SSP 30539

# Specification
# User Interface Language

February, 1992

The Charles Stark Draper Laboratory
Cambridge, Massachusetts

## TABLE OF CONTENTS

**TABLE OF CONTENTS (CONTINUED)**

## 1.0 INTRODUCTION

This Specification contains a detailed description of the syntax and semantics of the Space Station Freedom (SSF) User Interface Language (UIL).

The version of UIL specified in this document is based upon the TIMELINER language developed at the Charles Stark Draper Laboratory. To distinguish it from the UIL previously specified, the older version will be referred to as USE UIL.

### 1.1 History

The user interface language described in this document is based upon the TIMELINER language developed at the Charles Stark Draper Laboratory (CSDL) of Cambridge, Massachusetts.

This language was developed in 1980 to provide a tool for driving simulations. It met a need to initialize simulations, insert errors and perturbations, provide "human" inputs to simulate a crew, and recover information from the simulation. The TIMELINER language has been used by CSDL in a variety of simulation projects.

The initial version of the TIMELINER language was implemented using the HAL/S programming language. In about 1986 a Fortran-language version was created for use in a simulation of the Aerobrake Flight Experiment (AFE) spacecraft. In 1990 an Ada-language version was developed for use on the CSDL real-time test-bed of the SSF DMS.

Meanwhile, the need for a capable human-computer interface language was recognized by the Space Station Freedom Program (SSFP), and a specification describing such a language was developed. The last version of this specification is dated June, 1991.

For programmatic reasons the development of UIL was deferred. However, in the spring of 1991 the need was recognized for an "interim" version of UIL for use during the early stages of space station assembly.

TIMELINER provided this capability. In fact, as discussions continued during the summer of 1991, it became clear that TIMELINER, although more austere than UIL, provided almost all of the capability promised by the pre-existing UIL specification. It became apparent that TIMELINER could serve the needs of the SSF for the life of the program.

TIMELINER was selected for use on SSF by the Preliminary Software Control Board chaired by Mr. Richard Thorson, on September 27, 1991. This document was first prepared for reference by the change request document required to incorporate TIMELINER into the SSFP.

This specification therefore replaces the "old" UIL Specification document mentioned above. The language referred to as UIL in this document and elsewhere is the language that began its life at the Draper Laboratory as the TIMELINER language.

### 1.2 Scope of the language

The purpose of UIL is to provide a means of creating procedures that can provide sequencing and command functions for the SSF system. UIL procedures may be used to implement such activities as the "master timeline", "short term plans", payload operations, and so forth. Such procedures may be operated automatically, or operated under the monitoring of the onboard crew or ground controllers.

Note that the term "procedure" is used in this document to designate any UIL script. UIL scripts are grouped into "bundles", "sequences", and "subsequences", as described in Section 3.2 below.

The version of UIL documented here is designed for use in any computational environment that (1) can execute a "main program" written in the Ada programming language, and (2) has access to the SSF Data Management System (DMS) Standard Services (STSV) by means of the interfaces described in the Onboard Application Programming Interface Definition (APID).

Note that the UIL described in this document is "descoped" with respect to "old UIL". UIL was previously visualized as a common language that would be implemented separately for use in diverse environments, including but not limited to the onboard environment. In contrast, the current version is limited to systems that contain DMS services, and is based on an existing implementation.

Other versions of the TIMELINER language exist that contain the same functionality, but have interfaces allowing operation in different environments. Such versions are not covered by this document.

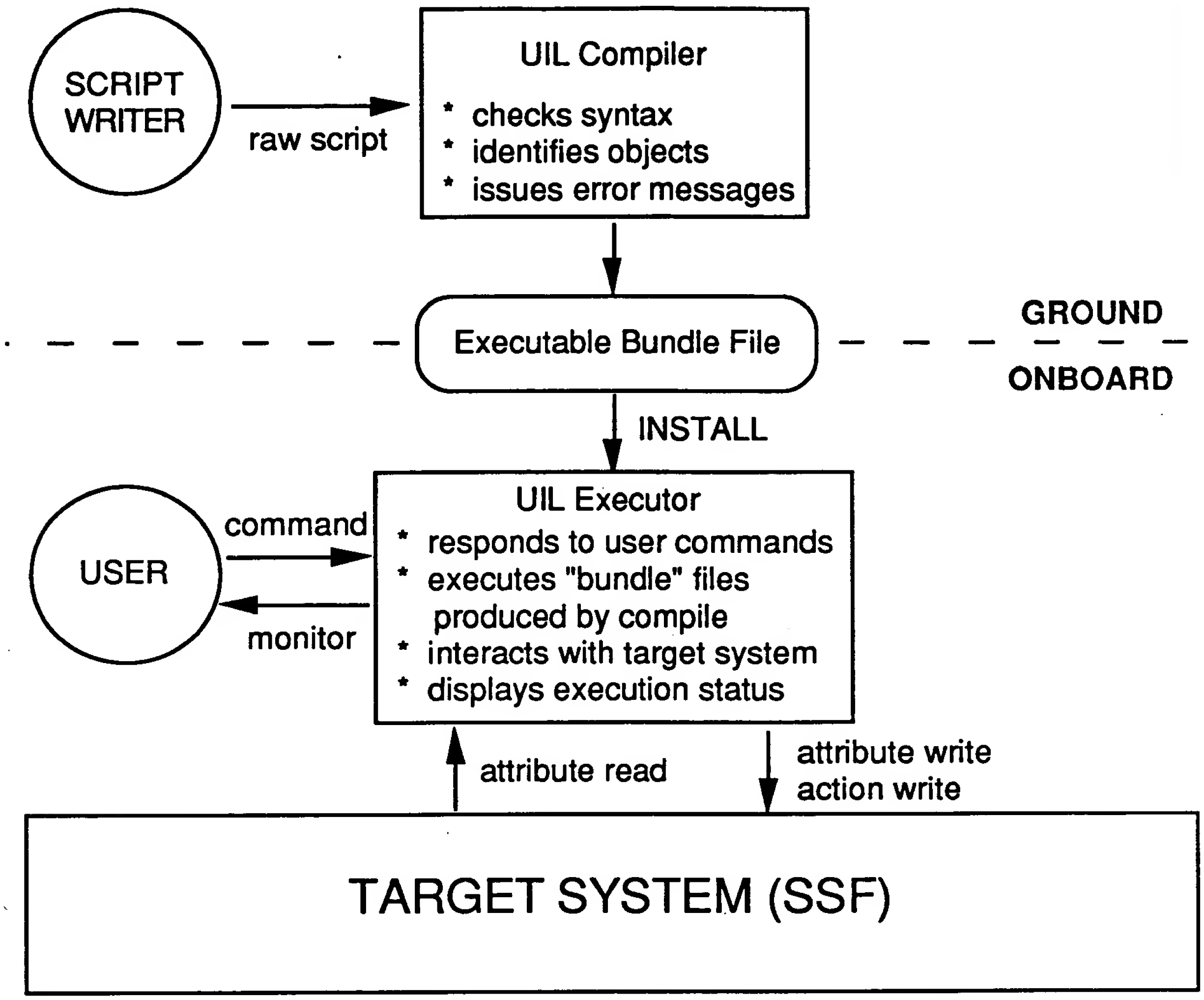
The exterior interfaces of UIL are by means of the "objects" defined in the Master Object Data Base (MODB), specifically those objects that form part of the Runtime Object Data Base (RODB) existing as part of the flight system. UIL's interfaces may be summarized as follows:

(1) The RODB interfaces needed to interact with the "target" system: These interfaces allow a UIL script to (a) read and make decisions based on RODB object attributes, (b) to write to object attributes, and (c) to command object actions.

(2) The RODB interfaces needed to support real-time user control of the execution of UIL procedures. These interfaces include (a) attributes that make visible information on the current status of the UIL Executor and the particular bundles that are currently resident, (b) actions that permit users to control the execution of UIL bundles and sequences, and (c) interfaces allowing UIL to retrieve executable files from a mass storage device.

## 1.3 Overview of the Implementation

This section provides an overview of the implementation of UIL that may be helpful in understanding the language specification. The following picture illustrates the major parts of the UIL implementation:



The implementation of UIL has two parts:

First, there must be a ground-based system for "compiling" bundles. At compile time the "raw" bundle is read in as ASCII data. The statements are parsed, identifiers are checked, and error messages are issued if necessary. The output of the compiler is a file of executable data. There is one such file for each bundle.

Second, there must be an onboard system for executing bundles. The onboard system may execute multiple bundles simultaneously. The onboard system interfaces with the target system, i.e. the SSF and its systems, by means of DMS standard services.

A bundle is brought into local memory for execution by means of the INSTALL command, and removed by means of the REMOVE command. The INSTALL and REMOVE commands may be invoked either by a UIL script, or externally by a ground or onboard user. Other commands are available for use is controlling the execution of bundles and their constituent sequences, as described below in Chapter 4.

## 2.0 RELATED DOCUMENTATION

This section lists documents relevant to this specification.

### 2.1 Applicable Documents

SSP 30000, Space Station Program Definition and Requirements, current revision.

SSP 30555, Level A Integrated Flight Software Architecture Requirements Document, current revision.

NASA-STD-3000, Space Station Freedom Man-Systems Integration Standards Volume IV, current revision.

### 2.2 Reference Documents

COL-MBER-100-TN-0659-01, New Concept for the Columbus User Control Language; MBB ERNO, current revision.

COL-MBER-000-TN-0341-00, Columbus Common MMI and User Control/Command Language; MBB ERNO, current revision.

JSC-09958, Space Shuttle Flight Data File Preparation Standards; Johnson Space Center, current revision.

JSC-22360, User Interface Language Operations Concepts - Preliminary; Debra A. Muratore, Mitre, 1986.

JSC-30497, Space Station Information System User Support Environment Functional Requirements; Johnson Space Center, current revision.

MTR-88-D0033-01, Dynamic Display of Crew Procedures for Space Station, Volume I - Baseline Definition; Gordon L. Johns, Mitre, 1988.

MTR-88-D0033-02, Dynamic Display of Crew Procedures for Space Station, Volume II - Functional Requirements; Gordon L. Johns, Mitre, 1988.

STD 1216 804, Columbus Human Computer Interface (HCI) Standard; MBB ERNO, current revision.

SSP 30261, Architectural Control Document, Data Management System, current revision.

## 3.0 LANGUAGE DEFINITION

This chapter defines the SSFP User Interface Language.

### 3.1 Notation

This section describes the notation that is used in this document to describe the UIL language:

(a) Reserved words including statement keywords are denoted by words printed in upper-case letters, for example: SEQUENCE, WHEN, AS.

(b) Syntactical elements including statement "components" are denoted by the use of a phrase in lower-case letters enclosed by corner brackets, for example: <name>, <singular_numeric>, <boolean_combo>. Such syntactical elements are described in Section 3.7.

(c) Square brackets are used to enclose optional elements. The elements within brackets may occur once at most.

(d) Braces (curly brackets) are used to enclose repeated elements. The elements within braces may appear zero or more times.

(e) A vertical bar ("|") separates alternative elements.

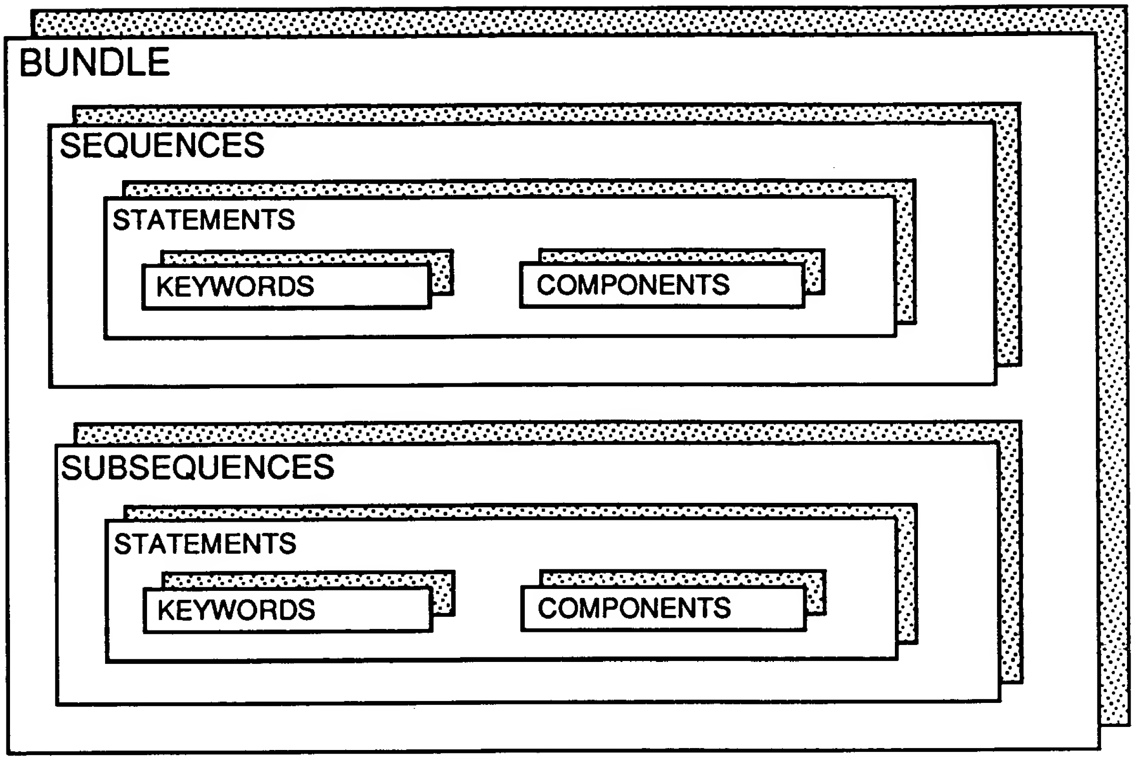
Note that UIL is case insensitive. That is, the compiler will accept upper-case, lower-case, or mixed forms of every reserved word; every RODB object, attribute, action or parameter name; every function or constant name; and every name created by a statement. The listing created at compile time, and all displays created during execution time, will use upper-case letters.

### 3.2 Language Hierarchy

UIL is a "procedural" language capable of executing "scripts".

The word "script" is used loosely to describe some UIL software entity that provides sequencing control. The elements used to build scripts are "bundles", "sequences" and "subsequences", and "statements". Bundles contain sequences and subsequences, which in turn contain statements. Statements are made up of "keywords" and "components".

The following picture illustrates the hierarchy that applies to UIL scripts:



The elements of the hierarchy are described in the following sections.

### 3.2.1 Bundle

The "bundle" is the uppermost hierarchical level of a UIL script. A bundle will normally be devoted to some particular, defined purpose that involves sequencing.

The "bundle" is defined as a set of one or more sequences. It may also contain one or more subsequences. The number of sequences and subsequences in a bundle is arbitrary, limited only by some implementation-dependent maximum. The user is free to establish any number of separate bundles, or sequences within a bundle, that is necessary to accomplish a given purpose.

The only kind of statement that may appear outside of a bundle in any compilation unit is the compiler directive statement (not described in this specification).

Except for DECLARE statements (Section 3.6.1) and DEFINE statements (Section 3.6.2), all the statements in any bundle must form part of one of its constituent sequences and subsequences.

The executable data for each bundle is packaged as a separate file by the UIL compiler. Such files will then be conveyed to the SSF by means of uplink, and will be stored in the Mass Storage Unit (MSU).

To be executed, the file corresponding to a bundle must be brought into the "local" memory of the processor where the UIL Executor resides. The bundle is the UIL unit that may be "installed" for execution, and "removed" when no longer needed. The INSTALL and REMOVE functions are discussed below in Chapter 4.

Note that a cyclic Ada task is established to process each bundle when it is installed. This task executes at a particular Ada priority and cycles at a particular delta-time (DT) interval. The priority and DT are specified as parameters to the INSTALL command. The priority and DT should be selected according to the importance of the particular bundle, and according to the time granularity at which the bundle must sense the occurrence of some event or condition.

The source code for a bundle must begin with a "bundle header" statement (Section 3.3.1) and conclude with a CLOSE statement (Section 3.3.4).

### 3.2.2 Sequence

A sequence establishes one and only one stream of execution. The stream of execution established by a sequence functions in parallel with the streams established by other sequences. The user may organize a script into any number of sequences as required to accomplish a given purpose.

A sequence contains a set of statements that will be executed serially. That is, they will be executed in their order or occurrence, except as modified by the operation of conditional constructs such as WHEN, WHENEVER, EVERY, and IF.

On a given pass, the active sequences in a given bundle will be processed in the order that they appear in the raw script.

A sequence must begin with a "sequence header" statement (Section 3.3.2) and conclude with a CLOSE statement (Section 3.3.4).

### 3.2.3 Subsequence

A subsequence, like a sequence, contains a set of statements that will be executed serially. However, a subsequence does not establish a new parallel stream of execution. The statements in a subsequence are processed when the subsequence is called (see Section 3.4.12).

Subsequences appear in a TIMELINER script at the same level as sequences. A subsequence must not be placed physically inside a sequence. A subsequence may be called by any sequence or subsequence within the same bundle.

Subsequence calls may be nested. However, a subsequence forms part of the stream of execution established by the sequence that makes the upper-level call. A subsequence that is never called will compile -- but cannot be executed.

A subsequence must begin with a "subsequence header" statement (Section 3.3) and conclude with a CLOSE statement (Section 3.3.4).

### 3.2.4 Statement

Bundles contain sequences and subsequences. Sequences and subsequences, in turn, contain statements.

Every UIL statement starts with a distinctive keyword -- a reserved word that specifies the statement type. A statement may continue onto multiple lines of the ASCII input to the compiler. Every statement must begin on a new line. The compiler recognizes the beginning of a new statement by encountering a keyword at the start of an input line.

UIL statements are of four types:

Blocking statements are statements used to name and mark the beginning and end of UIL bundles, sequences, and subsequences. Blocking statements are initiated by the following keywords, and further discussed in the referenced sections of this specification:

| | | | |
|---|---|---|---|
| BUNDLE | | | Section 3.3.1 |
| SEQUENCE | or | SEQ | Section 3.3.2 |
| SUBSEQUENCE | or | SUBSEQ | Section 3.3.3 |
| CLOSE | | | Section 3.3.4 |

Control statements include statements associated with the WHEN, WHENEVER, EVERY and IF constructs that are used to specify the conditions under which actions should occur. The WAIT and CALL statements are also considered to be "control" statements. Control statements are initiated by the following keywords, and further discussed in the referenced sections of this specification:

| | | | |
|---|---|---|---|
| WHEN | | | Section 3.4.1 |
| WHENEVER | | | Section 3.4.2 |
| EVERY | | | Section 3.4.3 |
| IF | | | Section 3.4.4 |
| BEFORE | | | Section 3.4.5 |
| WITHIN | | | Section 3.4.6 |
| OTHERWISE | | | Section 3.4.7 |
| ELSE | | | Section 3.4.8 |
| ELSEIF | or | ELSIF | Section 3.4.9 |
| END | | | Section 3.4.10 |
| WAIT | | | Section 3.4.11 |
| CALL | | | Section 3.4.12 |

Action statements stand alone in themselves and perform some "action". Action statements include the SET statement used to perform RODB object-attribute writes (and to set internal variables) and the COMMAND statement used to command RODB object-actions. UIL also includes dedicated START, STOP, and RESUME statements that can be used to control the execution of UIL sequences within a bundle. (the bundle and sequence control functions are available as RODB actions invoked by the COMMAND statement.) Action statements are initiated by the following keywords, and further discussed in the referenced sections of this specification:

| | |
|---|---|
| SET | Section 3.5.1 |
| COMMAND | Section 3.5.2 |
| START | Section 3.5.3 |
| STOP | Section 3.5.4 |
| RESUME | Section 3.5.5 |

Non-executable statements consist of the DEFINE statement used to create a name that points to a "component", and the DECLARE statement used to create internal variables. Compiler directives also exist, but are outside the scope of this document. Action statements are initiated by the following keywords, and further discussed in the referenced sections of this specification:

| | |
|---|---|
| DECLARE | Section 3.6.1 |
| DEFINE | Section 3.6.2 |

### 3.2.5 Component

Once the statement type is known, the compiler evaluates the remainder of the statement. Every statement consists of components and reserved words. UIL reserved words are described in Section 3.7.1 of this specification.

"Component" is a general term used to cover a variety of "things" that may occur in a UIL statement.

Components may be of three types: boolean, numeric, and character string.

Within each type, a component may be a literal, an RODB attribute, a list, a combination, an internal variable, a definition, a combination, or a built-in function. Several special-purpose components also exist, such as the "wild card", an asterisk that may be used in a subscript expression.

A third characteristic of components, that cuts across the two dimensions just described, is the size of the component. Component size is one-dimensional. Therefore, for example, UIL does not object if a 9-element array is assigned to a 3x3 array.

Component types are described more fully in Section 3.7.4 of this specification.

### 3.2.6 Comments and Blank Lines

UIL permits comments to be added to a script. Comments are visible in the raw script and in the compile-time listing of the script. As in any computer language, comments are not retained in the executable code.

Two or more adjacent hyphens (i.e. "--") mark the beginning of a comment. All material between the first double hyphen on a line and the end of the line is considered to be part of the comment. A comment may occupy a line by itself, or share a line with a statement.

Blank lines are legal and the spacing implied by blank lines is retained in the compile-time listing of the script.

## 3.3 Blocking Statements

This chapter describes the "blocking" statements that are used in the source code of UIL bundles to mark the beginning and end of UIL bundles, sequences, and subsequences.

### 3.3.1 The BUNDLE Header Statement

The uppermost hierarchical level of a UIL script is the "bundle". The beginning of a bundle is marked by a header statement of the form:

```
      BUNDLE <name>
```

where <name> is an alphanumeric word without imbedded blanks chosen by the user. (See Section 3.7.2)

Each bundle is closed by a CLOSE BUNDLE statement, as described below in Section 3.3.4.

### 3.3.2 The SEQUENCE Header Statement

Multiple sequences, which execute in parallel with each other, contain the substance of a UIL bundle. The beginning of a sequence is marked by a header statement of the form:

```
      SEQ[UENCE]  <name>  [ACTIVE | INACTIVE]
```

where <name> is an alphanumeric word without imbedded blanks chosen by the user. (See Section 3.7.2)

An optional third word may be added to a SEQUENCE header statement to specify the initial state of the sequence. ACTIVE means that the sequence will start executing as soon as the parent bundle is installed. INACTIVE means that the sequence will be stopped before executing its first statement. The default is ACTIVE.

Each sequence is closed by a CLOSE SEQUENCE statement, as described below in Section 3.3.4.

### 3.3.3 The SUBSEQUENCE Header Statement

Subsequences are like sequences except that they form part of the stream of execution of the sequence that calls the subsequence. The beginning of a subsequence is marked by a header statement of the form:

```
      SUBSEQ[UENCE]  <name>
```

where <name> is an alphanumeric word without imbedded blanks chosen by the user. (See Section 3.7.2)

Each subsequence is closed by a CLOSE SUBSEQUENCE statement, as described below in Section 3.3.4.

### 3.3.4 The CLOSE Statement

The CLOSE statement is required to mark the end of UIL bundles, sequences, and subsequences.

A bundle is ended by the statement

```
      CLOSE  BUNDLE  [<name>]
```

When compile-time processing reaches a CLOSE BUNDLE statement the file containing the executable code for the bundle is closed. A new file will be opened if the compilation unit contains additional bundles. This statement has no function during execution-time.

A sequence is ended by the statement

```
      CLOSE  SEQ[UENCE]  [<name>]
```

When execution-time processing reaches a CLOSE SEQUENCE statement the sequence in question is concluded and placed in an inactive state.

A subsequence is ended by the statement

```
      CLOSE  SUBSEQ[UENCE]  [<name>]
```

When execution-time processing reaches a CLOSE SUBSEQUENCE statement processing returns to the sequence or subsequence containing the CALL statement that invoked the subsequence, at the statement immediately following the CALL statement.

The CLOSE statement may contain the name of the block being closed. In the event that a <name> is provided, a compilation error message will be issued if it does not match the name given by the corresponding header statement.

## 3.4 Control Statements

This chapter describes the "control" statements that permit the UIL user to specify the conditions under which actions are to occur.

Some UIL control statements initiate "constructs" -- namely the WHEN, WHENEVER, EVERY and IF statements. Such constructs are concluded by an END statement. Other control statements, such as BEFORE, WITHIN, OTHERWISE, ELSE, and ELSE IF, are used within these constructs. Two additional, stand-alone statements are classified as control statements, namely WAIT and CALL.

### 3.4.1 The WHEN Construct

The WHEN statement is used to specify conditions at which an action is to occur -- on a one-shot basis. The WHEN statement initiates a WHEN construct.

All forms of the WHEN construct contain a "condition" expressed as a <singular_boolean> component -- that is, an unarrayed component that can be evaluated as TRUE or FALSE. UIL components are discussed below in Section 3.7.4.

#### 3.4.1.1 The One-Line WHEN Construct

The simplest form of the WHEN construct consists of a single statement of the form:

```
      WHEN <singular_boolean> [THEN] CONTINUE
```

This statement, when encountered in a given sequence during execution, causes the sequence to pause until the condition expressed as a <singular_boolean> component is true. When the condition is met execution continues with the next statement.

#### 3.4.1.2 The Simple WHEN Construct

The WHEN construct may also be written as follows:

```
      WHEN  <singular_boolean>  [THEN]
            <statements>
      END  [WHEN]
```

The WHEN statement, when encountered in a given sequence during execution, causes the sequence of which it is a part to pause until the condition expressed as a <singular_boolean> component is true. When the condition is fulfilled, the statements following the WHEN statement are processed until the END statement is encountered. Processing then continues at the statements following the END statement.

Note that this form of the WHEN construct is equivalent to the first form. It may however be desirable when the enclosed statements are more intimately connected to the condition than the statements that would follow the END statement, or when the user anticipates that a modifier statement such as BEFORE or WITHIN will later be added.

#### 3.4.1.3 The Modified WHEN Construct

Additional forms of the WHEN construct may be created by use of the BEFORE and WITHIN statements. Furthermore, if WHEN is modified by BEFORE or WITHIN, an OTHERWISE statement may be used.

The modified form of the WHEN construct is as follows:

```
      WHEN  <singular_boolean>
         [BEFORE  <singular_boolean>  |  WITHIN  <singular_numeric>]
            <statements>
         [OTHERWISE
            <statements>]
      END  [WHEN]
```

The WHEN statement, when encountered, causes the sequence of which it is a part to pause until the condition expressed as a <singular_boolean> component is true. When the condition is met, the statements following the BEFORE or WITHIN statement are processed until either an OTHERWISE or an END statement is encountered. Processing then continues at the statements following the END statement.

However, each time the WHEN condition is to be checked, the condition specified by the BEFORE or WITHIN statement is evaluated first. Regardless of whether the WHEN condition is fulfilled simultaneously, if the condition that forms part of the BEFORE statement is fulfilled, or if the time interval specified in the WITHIN statement has elapsed since the WHEN statement was encountered, then processing continues at the statements following the OTHERWISE statement. If there is no OTHERWISE statement, processing skips straight to the END statement and continues at the statements following the END.

In other words, the BEFORE and WITHIN statement may be used to terminate operation of the WHEN construct if a condition (BEFORE) is met, or if a time interval (WITHIN) elapses. An OTHERWISE statement allows a separate set of statements to be provided that will be processed when the construct is terminated.

### 3.4.2 The WHENEVER Construct

The WHENEVER statement is used to specify conditions at which an action is to occur -- on a repeating basis. The WHENEVER statement initiates a WHENEVER construct.

All forms of the WHENEVER construct contain a "condition" expressed as a <singular_boolean> component -- that is, an unarrayed component that can be evaluated as TRUE or FALSE. UIL components are described below in Section 3.7.4.

### 3.4.2.1 The Simple WHENEVER Construct

The simple form of the WHENEVER construct is as follows:

```
WHENEVER  <singular_boolean>  [THEN]
     <statements>
END  [WHENEVER]
```

The WHENEVER statement, when encountered, causes the sequence of which it is a part to pause until the condition expressed as a <singular_boolean> component is true. When the condition is met, the statements following the WHENEVER statement are processed until the END statement is encountered. Then processing returns to the WHENEVER statement. Upon the next off-to-on transition of the condition stated in the WHENEVER statement the interior statements are again processed. This loop repeats indefinitely.

Note that if the <singular_boolean> component transitions back to FALSE, and back to TRUE, during the interval between successive iterations of the bundle, or during the time that the interior statements are being processed, these transitions will not be sensed by the WHENEVER construct. The construct will act as though the condition were continuously present.

### 3.4.2.2 The Modified WHENEVER Construct

The modified form of the WHENEVER construct is as follows:

```
WHENEVER  <singular_boolean>  [THEN]
   [BEFORE  <singular_boolean>  |  WITHIN  <singular_numeric>]
      <statements>
END  [WHENEVER]
```

The WHENEVER statement, when encountered, causes the sequence of which it is a part to pause until the condition expressed as a <singular_boolean> component is true. When the condition is met, the statements following the BEFORE or WITHIN statement are processed until the END statement is encountered. Then processing returns to the WHENEVER statement. The loop repeats upon the next off-to-on transition of the WHENEVER condition.

However, each time the WHENEVER condition is to be checked, the condition established by the BEFORE or WITHIN statement is first evaluated. Regardless of whether the WHENEVER condition is fulfilled, if the condition that forms part of the BEFORE statement is fulfilled, or if the time interval specified in the WITHIN statement has elapsed since the WHENEVER statement was first encountered, then processing skips straight to the END statement and then drops through to the statements that follow the END statement.

In other words, the BEFORE and WITHIN statement may be used to terminate the loop created by the WHENEVER statement if a (BEFORE) condition is met, or if a (WITHIN) time interval elapses. Without a BEFORE or WITHIN statement a WHENEVER construct will loop indefinitely.

### 3.4.3 The EVERY Construct

The EVERY statement is used to make some actions occur repetitively at some specified time interval. The EVERY statement initiates an EVERY construct.

All forms of the EVERY construct contain a time interval expressed as a <singular_numeric> component -- that is, an unarrayed component that can be evaluated as a number of seconds. UIL components are discussed below in Section 3.7.4. The time interval may be expressed as a literal in time format as described in Section 3.7.4.1.2.

### 3.4.3.1 The Simple EVERY Construct

The simple form of the EVERY construct is as follows:

```
EVERY  <singular_numeric>  [THEN]
      <statements>
END  [EVERY]
```

The EVERY statement, when first encountered, immediately drops through to execute the statements that lie between the EVERY statement and the END statement. When the END statement is reached, processing returns to the EVERY statement. When the time interval expressed as a <singular_numeric> component has elapsed, since the previous encounter with the EVERY statement, the interior statements are again processed. This loop repeats indefinitely.

Note that the accuracy with which the UIL executor can match the specified time interval is related to the delta-time at which the parent task is being processed. The fastest possible iteration of the EVERY loop is that interval equal to the delta-time of the parent bundle. Any EVERY loop whose interval is less than the parent delta-time will be processed at the delta-time iteration rate. Furthermore, if the execution of the enclosed statements takes a time longer than the interval expressed by the <singular_numeric> component, the prescribed time for the next iteration will be missed.

### 3.4.3.2 The Modified EVERY Construct

The modified form of the EVERY construct is as follows:

```
      EVERY  <singular_numeric>  [THEN]
         [BEFORE  <singular_boolean>  |  WITHIN  <singular_numeric>]
            <statements>
      END  [EVERY]
```

The EVERY statement, when first encountered, immediately drops through to execute the statements that lie between the BEFORE or WITHIN statement and the END statement. When the END statement is reached, processing returns to the EVERY statement. When the time interval specified by the EVERY statement has elapsed, the interior statements are again processed.

However, on each pass the condition established by the BEFORE or WITHIN statement is evaluated. If the condition that forms part of the BEFORE statement is fulfilled, or if the time interval specified in the WITHIN statement has elapsed since the EVERY statement was first encountered, then processing skips straight to the END statement and then drops through to the statements that follow the END statement.

In other words, the BEFORE and WITHIN statement may be used to terminate the loop created by the EVERY statement if a (BEFORE) condition is met, or if a (WITHIN) time interval elapses. Without a BEFORE or WITHIN statement an EVERY construct will loop indefinitely.

### 3.4.4 The IF Construct

The IF statement is used to choose among actions -- at a particular point in time. The IF statement initiates an IF construct.

Unlike the WHEN, WHENEVER, and WAIT constructs, an IF construct is not tied in any way to the passage of time. The choice embodied in the IF statement is processed all at once.

All forms of the IF construct contain a "condition" expressed as a <singular_boolean> component -- that is, an unarrayed component that can be evaluated as TRUE or FALSE. UIL components are discussed below in Section 3.7.4.

An IF construct in its simplest form consists of an IF statement, an END statement, and the statements that lie between. An IF construct may optionally contain multiple ELSE IF clauses and an ELSE clause.

The form of the IF construct is as follows:

```
IF  <singular_boolean>  [THEN]
     <statements>
{ELSEIF  <singular_boolean>  [THEN]
     <statements>}
[ELSE
     <statements>]
END  [IF]
```

When the IF statement is encountered, the condition expressed as a <singular_boolean> component is evaluated. If the condition passes, the statements that follow the IF statement are processed, and execution then skips to the END statement and falls through to whatever statements follow. If the condition fails, and there is no ELSEIF or ELSE clause, execution skips straight to the END statement.

If there are any ELSEIF or ELSE clauses, they are processed as follows: If the IF condition fails, execution skips to the first ELSEIF or ELSE. If it is an simple ELSE clause the statements lying between the ELSE and the END statement are processed. If it is an ELSEIF clause, the specified condition is evaluated and if true the statements lying between that ELSEIF and the next ELSEIF or ELSE or END are processed. This process continues until the IF construct is completed.

### 3.4.5 The BEFORE Statement

The BEFORE statement is a "modifier" used to create an alternative condition as part of a WHEN (3.4.1.2), WHENEVER (3.4.2.2), or EVERY (3.4.3.2) construct. Please consult the referenced sections of this Specification for an explanation of how BEFORE is used.

The form of the BEFORE statement is:

```
BEFORE  <singular_boolean>  [THEN]
```

where <singular_boolean> is an unarrayed component that may be evaluated as TRUE or FALSE. UIL components are described below in Section 3.7.4.

### 3.4.6 The WITHIN Statement

The WITHIN statement is a "modifier" used to create an alternative condition as part of a WHEN (3.4.1.2), WHENEVER (3.4.2.2), or EVERY (3.4.3.2) construct. Please consult the referenced sections of this Specification for an explanation of how WITHIN is used.

The form of the WITHIN statement is:

```
WITHIN  <singular_numeric>  [THEN]
```

where <singular_numeric> is an unarrayed numeric component that expresses a time interval in seconds. UIL components are described below in Section 3.7.4. The time interval may be expressed as a literal in time format as described in Section 3.7.4.1.2.

### 3.4.7 The OTHERWISE Statement

The OTHERWISE statement is used as part of a modified WHEN construct to introduce statements that are executed instead of the normal statements in the event that the BEFORE or WITHIN condition arises before or at the same time as the condition given in the WHEN statement itself, as described in Section 3.4.1.3 of this Specification.

The OTHERWISE statement is as follows:

```
OTHERWISE
```

A WHEN construct may contain at most one OTHERWISE statement.

### 3.4.8 The ELSE Statement

The ELSE statement is used as part of an IF construct to introduce statements to be executed if the original IF condition, or preceding ELSEIF conditions, fail -- as described in Section 3.4.4 above.

The ELSE statement is as follows:

```
ELSE
```

A particular IF construct may contain at most one ELSE statement.

### 3.4.9 The ELSEIF Statement

The ELSEIF statement is used as part of an IF construct to introduce statements to be executed if the preceding IF or ELSEIF condition(s) fail -- as described in Section 3.4.4 above.

The ELSEIF statement is as follows:

```
ELSEIF  <singular_boolean>  [THEN]
```

where <singular_boolean> is any unarrayed component that may be evaluated as TRUE or FALSE.

A particular IF construct may contain any number of ELSEIF statements.

### 3.4.10 The END Statement

The END statement is required to mark the end of control constructs, namely the WHEN (3.4.1), WHENEVER (3.4.2), EVERY (3.4.3), and IF (3.4.4) constructs. Please consult the referenced sections of this User's Guide for an explanation of how END is used.

The form of the END statement is:

```
      END  [<construct_type>]
```

where <construct_type> is the single word WHEN, WHENEVER, EVERY, or IF. The <construct_type>, if included, must agree with the type of the construct being concluded.

### 3.4.11 The WAIT Statement

The WAIT statement is used to introduce a pause into a UIL sequence. Only the particular sequence (or subsequence) containing the WAIT statement pauses.

The WAIT statement is of the form:

```
      WAIT  <singular_numeric>
```

where <singular_numeric> is an unarrayed numeric component that expresses a time interval in seconds. If the time interval is a variable, it will be evaluated only once, at the beginning of the wait. The time interval may be expressed as a literal in time format as described in Section 3.7.4.1.2. If the time interval is zero or negative, no pause will occur.

### 3.4.12 The CALL Statement

The CALL statement is used to invoke a subsequence, as described in Section 2.3 above. A CALL statement may be present in any UIL sequence or subsequence within a bundle, and may be nested within a control construct.

The CALL statement is of the form:

```
      CALL  <name>
```

When encountered, a CALL statement causes control to be transferred to the SUBSEQUENCE header statement that begins the subsequence that bears the specified <name>. The subsequence then executes. When the CLOSE statement that concludes the subsequence, is encountered, control is returned to the sequence or subsequence that contains the CALL statement. Execution then continues at the statement following the CALL statement.

A called subsequence must lie within the same UIL bundle as the sequence or subsequence that contains the CALL statement.

A control construct such as WHEN, WHENEVER, EVERY or IF must be completed within the same sequence or subsequence where it begins. That is, a control construct may not begin in the calling sequence (or subsequence) and end in the called subsequence, or vice versa.

An execution error will occur upon execution of a CALL statement if the depth of subsequence nesting exceeds an implementation-dependent maximum. If such an error occurs the sequence will be deactivated.

### 3.5 Action Statements

This chapter describes the "action" statements that specify the actions to be taken under conditions specified by UIL control statements.

Note that the START, STOP, and RESUME statements described below provide capabilities that overlap the user interface capabilities described in Chapter 4. The dedicated START, STOP and RESUME statements may be used only to control sequences within a given bundle. These statements do not require use of DMS services. In contrast, the user interface commands described in Chapter 4 do require use of DMS services, but they are more general. They may be invoked to control sequences in other bundles, or even in other instantiations of the UIL executor. The user interface capabilities are available within UIL sequences by means of the COMMAND statement (Section 3.5.2).

#### 3.5.1 The SET Statement

The SET statement is used to write to an RODB object attribute, or to set an internal variable.

The SET statement is of the form:

```
      SET  <target>  TO  <value>
```

where <target> is either an RODB <object_attribute> component or a <internal_variable> component, and <value> is any component that can be evaluated to provide a value of size (number of components) and type (boolean, numeric, or character string) that matches the <target>.

When encountered, the value will first be evaluated. Then the value will be written to the target.

If <target> is a variable created by a UIL DECLARE statement (Section 3.6.1) the setting will occur almost instantaneously, and processing will proceed to the next statement within the same iteration of the bundle.

If <target> is an RODB <object_attribute> DMS standard services will be used to convey the value to the attribute. In this case processing will not proceed to the next statement until DMS services indicate that the object-attribute has been set.

### 3.5.2 The COMMAND Statement

The COMMAND statement is used to write an RODB object action, with accompanying parameter(s) if required.

The COMMAND statement is of the form:

```
      COMMAND  <action>  <object>  {, <parameter> => <value>}
```

where <action> is a legal action with respect to the named <object>, <parameter> names a valid parameter for the object and action, and <value> is a component that can be evaluated to provide a value of size and type that is compatible with the named parameter.

### 3.5.3 The START Statement

The START statement is used to activate a sequence that lies within the same bundle.

The START statement is of the form:

```
      START  <name>
```

where <name> is an alphanumeric word that corresponds to the name on the SEQUENCE header statement of another sequence within the same bundle.

If the named sequence is presently inactive, the START statement causes its statement pointer to be reinitialized to the first statement in the sequence and the sequence is then activated. If the named sequence is currently active the START statement will have no effect.

Note that the function of the START statement may also be invoked by means of the user interface START command described in Section 4.1.5. The dedicated START statement described here does not use the DMS services, and can be used only to start a sequence within the same bundle.

### 3.5.4 The STOP Statement

The STOP statement is used to deactivate a sequence that lies within the same bundle.

The STOP statement is of the form:

```
      STOP  <name>
```

where <name> is an alphanumeric word that corresponds to the name on the SEQUENCE header statement of another sequence within the same bundle.

If the named sequence is presently active, the STOP statement causes it to be deactivated. If the named sequence is currently inactive the STOP statement will have no effect.

Note that the function of the STOP statement may also be invoked by means of the user interface STOP command described in Section 4.1.6. The dedicated STOP statement described here does not use the DMS services, and can be used only to stop a sequence within the same bundle.

### 3.5.5 The RESUME Statement

The RESUME statement is used to activate a sequence that lies within the same bundle. RESUME differs from START by starting the sequence wherever it was stopped, rather than starting it at its first statement.

The RESUME statement is of the form:

```
RESUME  <name>
```

where <name> is an alphanumeric word that corresponds to the name on the SEQUENCE header statement of another sequence within the same bundle.

If the named sequence is presently inactive, the RESUME statement causes it to be activated. The sequence's statement pointer is not changed, so the first statement executed will be the statement at which the sequence was stopped. If the named sequence is currently active the RESUME statement will have no effect.

Note that the function of the RESUME statement may also be invoked by means of the user interface RESUME command described in Section 4.1.7. The dedicated RESUME statement described here does not use the DMS services, and can be used only to RESUME a sequence within the same bundle.

### 3.5.6 The MESSAGE Statement

the MESSAGE statement saves character string information provided at compilation such that it can be displayed during execution. This capability can be used to supply informative material or to request manual actions.

The MESSAGE statement is of the form:

```
MESSAGE <cstring_lit>
```

where <cstring_lit> is a character string enclosed by double quotation marks.

In existing implementations the character information provided in a MESSAGE statement is made available during execution by printing it as part of the statement by statement output that is provided. The method by which such information can be made known to the onboard crew of the SSF is not known at this time.

## 3.6 Non-Executable Statements

Several non-executable statements are available. These statements have a "scope". That is, non-executable statements placed at the top of a bundle have an effect that extends across all the sequences and subsequences within the bundle. Non-executable statements placed at the top of a sequence or subsequence have effect only within the sequence or subsequence.

### 3.6.1 The DECLARE Statement

The DECLARE statement is used to create an internal variable.

The DECLARE statement is of the form

```
      DECLARE  <name>  <BOOLEAN | NUMERIC | CHARACTER>[(<num_ntgr_lit>)]
```

where <name> is an alphanumeric word without embedded blanks, and <num_ntgr_lit> is a numeric literal expressed as an integer.

The DECLARE statement creates and allocates an internal variable of boolean, numeric, or character string type. The variable may be arrayed in at most one dimension. If the variable is arrayed, the number of its elements is designated by the <num_ntgr_lit> component enclosed in parentheses. The subscripts for an arrayed internal variable always begin with one.

Examples of the DECLARE statement are

```
      DECLARE  XYZ_FLAG  BOOLEAN          -- singular boolean
      DECLARE  R_VECTOR  NUMERIC(3)       -- array of 3 numerics
      DECLARE  HOC_NAME  CHARACTER(32)    -- string of 32 characters
```

A DECLARE statement must be placed before the first executable statement in a bundle, sequence, or subsequence. The scope of the DECLARE statement depends upon its placement. A DECLARE placed at the top of a bundle creates a name that is visible within all the sequences and subsequences of the bundle. A DECLARE statement placed at the top of a particular sequence or subsequence creates a name that is visible throughout that sequence or subsequence.

The DECLARE statement does not permit internal variables to be initialized to an arbitrary value. All boolean variables are initialized to FALSE, all numeric variables are initialized to zero, and all character variables are initialized to blanks. Variables are initialized to these values whenever the bundles containing their DECLARE statements are installed.

### 3.6.2 The DEFINE Statement

The DEFINE statement is used to create a name that references some other component.

The DEFINE statement is of the form

```
      DEFINE  <name>  AS  <component>
```

where <name> is an alphanumeric word without embedded blanks, and <component> is any legal component.

Examples of the DEFINE statement are

```
DEFINE  KU_POWER  AS  POWER_STATUS OF KU_SYSTEM
DEFINE  ALTITUDE  AS  R_MAGNITUDE - EARTH_RADIUS
```

The DEFINE statement creates a name. Within the scope of the statement, any reference to the name has the same effect as a reference to the component to which the name refers.

A DEFINE statement must be placed before the first executable statement in a bundle, sequence, or subsequence. The scope of the DEFINE statement depends upon its placement. A DEFINE placed at the top of a bundle creates a name that is visible within all the sequences and subsequences of the bundle. A DEFINE statement placed at the top of a particular sequence or subsequence creates a name that is visible throughout that sequence or subsequence.

A DEFINE statement may reference an internal variable. However, the DECLARE statement for the variable must precede the DEFINE statement.

## 3.7 Syntactical Elements

This chapter describes the syntactical elements that are available for forming statements. Such syntactical elements include reserved words, operators, and components.

### 3.7.1 Reserved Words

This section summarizes all the reserved words that go into the formation of statements and components. For information on how each reserved word is used see the section whose number is given in parentheses.

| | |
|---|---|
| BUNDLE | blocking statement type (3.3.1) |
| SEQ | blocking statement type (3.3.2) |
| SEQUENCE | blocking statement type (3.3.2) |
| SUBSEQ | blocking statement type (3.3.3) |
| SUBSEQUENCE | blocking statement type (3.3.3) |
| CLOSE | blocking statement type (3.3.4) |
| WHEN | control statement/construct type (3.4.1) |
| WHENEVER | control statement/construct type (3.4.2) |
| EVERY | control statement/construct type (3.4.3) |
| IF | control statement/construct type (3.4.4) |
| BEFORE | control statement type (3.4.5) |
| WITHIN | control statement type (3.4.6) |
| OTHERWISE | control statement type (3.4.7) |
| ELSIF | control statement type (3.4.8) |
| ELSEIF | control statement type (3.4.8) |
| ELSE | control statement type (3.4.9) |
| END | control statement type (3.4.10) |
| WAIT | control statement type (3.4.11) |
| CALL | control statement type (3.4.12) |
| CONTINUE | WHEN statement keyword (3.4.1.1) |
| THEN | optional keyword (3.4.1 - 3.4.4) |

| | |
|---|---|
| SET | action statement type (3.5.1) |
| TO | SET statement keyword (3.5.1) |
| COMMAND | action statement type (3.5.2) |
| START | action statement type (3.5.3) |
| STOP | action statement type (3.5.4) |
| RESUME | action statement type (3.5.5) |
| COMMENT | action statement type (3.5.6) |
| DECLARE | non-executable statement type (3.6.1) |
| BOOLEAN | DECLARE-statement keyword (3.6.1) |
| NUMERIC | DECLARE-statement keyword (3.6.1) |
| CHARACTER | DECLARE-statement keyword (3.6.1) |
| DEFINE | non-executable statement type (3.6.2) |
| AS | DEFINE-statement keyword (3.6.2) |
| AND | logical operator (3.7.3.1) |
| OR | logical operator (3.7.3.1) |
| NOT | logical operator (3.7.3.1) |
| IN | range operator (3.7.3.7) |
| OUTSIDE | range operator (3.7.3.7) |
| MOD | arithmetic operator (3.7.3.4) |
| OF | object-attribute keyword (3.7.4.2) |
| ON | boolean literal (3.7.4.1.1) |
| OFF | boolean literal (3.7.4.1.1) |
| TRUE | boolean literal (3.7.4.1.1) |
| FALSE | boolean literal (3.7.4.1.1) |
| DEG_TO_RAD | built-in constant (3.7.4.4) |
| RAD_TO_DEG | built-in constant (3.7.4.4) |
| FT_TO_M | built-in constant (3.7.4.4) |
| M_TO_FT | built-in constant (3.7.4.4) |
| FT_TO_NM | built-in constant (3.7.4.4) |
| NM_TO_FT | built-in constant (3.7.4.4) |
| G_TO_FPS2 | built-in constant (3.7.4.4) |
| FPS2_TO_G | built-in constant (3.7.4.4) |
| LBM_TO_KG | built-in constant (3.7.4.4) |
| KG_TO_LBM | built-in constant (3.7.4.4) |
| SLUG_TO_KG | built-in constant (3.7.4.4) |
| KG_TO_SLUG | built-in constant (3.7.4.4) |
| LBF_TO_N | built-in constant (3.7.4.4) |
| N_TO_LBF | built-in constant (3.7.4.4) |
| ABS | built-in function (3.7.4.5) |
| SQRT | built-in function (3.7.4.5) |
| SIN | built-in function (3.7.4.5) |
| COS | built-in function (3.7.4.5) |
| TAN | built-in function (3.7.4.5) |
| ARCSIN | built-in function (3.7.4.5) |
| ARCCOS | built-in function (3.7.4.5) |
| ARCTAN | built-in function (3.7.4.5) |

### 3.7.2 Names

Names are created by the following statements:

| | |
|---|---|
| BUNDLE | name of bundle (Section 3.3.1) |
| SEQUENCE | name of sequence (Section 3.3.2) |
| SUBSEQUENCE | name of subsequence (Section 3.3.3) |
| DECLARE | name of internal variable (Section 3.6.1) |
| DEFINE | name of defined component (Section 3.6.2) |

Names are a string of characters chosen by the writer of the script. Names must be unique within their scope. Bundle names should be unique within the set of bundles available onboard. Sequence and subsequence names, and names created by DECLARE or DEFINE statements appearing at the top of a bundle must be unique within the bundle. Names created by DECLARE and DEFINE statements within a sequence or subsequence must be unique within their sequence or subsequence.

Names must begin with a letter of the alphabet, and must avoid using any symbols that are used to form operators (see Section 3.7.3). Names may not contain blanks, but underscores are permitted. Names are limited in length to an implementation-dependent maximum, which is currently set at 32 characters. A name may not be a reserved word.

### 3.7.3 Operators

This section summarizes the operators that are used in the formation of components. Somewhat arbitrarily the operators are grouped into several categories.

The order of precedence of operators is as follows, ranging from the operators that create the loosest associations to those that create the tightest:

| | | |
|---|---|---|
| AND | (Note 1) | logical operator (3.7.3.1) |
| OR | (Note 1) | logical operator (3.7.3.1) |
| = | | equality operator (3.7.3.2) |
| /= | | equality operator (3.7.3.2) |
| < | | comparison operator (3.7.3.3) |
| <= | | comparison operator (3.7.3.3) |
| > | | comparison operator (3.7.3.3) |
| >= | | comparison operator (3.7.3.3) |
| , | | list operator (3.7.3.5) |
| IN | (Note 1) | range operator (3.7.3.6) |
| OUTSIDE | (Note 1) | range operator (3.7.3.6) |
| .. | | range operator (3.7.3.6) |
| + | (Note 2) | arithmetic operator (3.7.3.4) |
| - | (Note 2) | arithmetic operator (3.7.3.4) |
| * | | arithmetic operator (3.7.3.4) |
| / | | arithmetic operator (3.7.3.4) |
| MOD | (Note 1) | arithmetic operator (3.7.3.4) |
| ** | | arithmetic operator (3.7.3.4) |
| NOT | (Note 1) | logical operator (3.7.3.1) |
| + | (Note 3) | arithmetic (sign) operator (3.7.3.4) |
| - | (Note 3) | arithmetic (sign) operator (3.7.3.4) |
| OF | | RODB operator (3.7.3.7) |
| => | | RODB operator (3.7.3.7) |

Note 1: This operator will be recognized only if it is preceded by a blank or a close-parenthesis, and anteceded by a blank or open-parenthesis.

Note 2: If preceded by an "E" and anteceded by a numeral the plus and minus symbols are interpreted as forming part of an exponent attached to a numeric literal.

Note 3: If the parser finds no components to their left, the plus and minus symbols are interpreted as unary operators that modify the numeric component lying to their right.

### 3.7.3.1 Logical Operators

The UIL operators AND, OR, and NOT are known as "logical" operators. These operators operate only on boolean components.

*NOT*

The NOT operator is a unary operator that modifies the sense of the singular or plural boolean component lying to its right. The result of the operator is a component of equal size with the logical sense of each element reversed.

If the NOT operator has any component to its immediate left (with no other operator in between) an error message will be issued at compile time.

Here are examples of proper and improper uses of the NOT operator:

```
NOT (ALT = 12345)     ==>legal
ALT/=12345            ==>legal
ALT NOT = 12345       ==>illegal
```

*AND, OR*

The AND and OR operators are binary operators that create a component of <boolean_combo> type (Section 3.7.4.6.1) by making the appropriate logical combination of the components to the left and right.

The components to the left and right must both be of boolean type and a compiler error is generated if they are not.

The components to the left and right of the operator must be compatible in size. If the size is equal the resulting component will be the appropriate logical combination of the corresponding elements of the left and right components. If one component is singular and the other has more than one element, the resulting plural component is the appropriate logical combination of each element of the plural component with the singular component. If both components are plural, but their size is not equal, an error message is generated by the compiler.

For example

```
(ON, OFF, ON)  AND  (ON, ON, OFF)   ==>  (ON, OFF, OFF)
(ON, OFF, ON)  OR  OFF              ==>  (ON, OFF, ON)
(ON, OFF, ON)  AND  (ON, OFF)       ==>  illegal
```

A compiler error is generated if the components to the left and right of the AND or OR operator are both plural, but have an unequal number of elements.

### 3.7.3.2 Equality Operators

The UIL equality operators are as follows:

| | |
|---|---|
| = | equals |
| /= | not equals |

The equality operators are binary operators that create a component of <boolean_combo> type (Section 3.7.4.6.1). The resulting component is always singular.

The components to the left and right of an equality operator must be of the same type, whether boolean, numeric, or character. A compiler error is generated if they are not.

The "=" operator returns the value TRUE if every element of the component on one side equals the corresponding element on the other side. Otherwise FALSE is returned.

If one of the components being combined is plural and the other is singular, TRUE is returned if every element of the plural component is equal to the singular component. Otherwise FALSE is returned.

If the components to the left and right are both plural, but of unequal sizes, an error message is generated by the compiler.

The "/=" operator returns the reverse of what the "=" operator would return.

### 3.7.3.3 Comparison Operators

The UIL comparison operators are as follows:

| | |
|---|---|
| < | less than |
| <= | less than or equals |
| > | greater than |
| >= | greater than or equals |

The comparison operators are binary operators that create a component of <boolean_combo> type (Section 3.7.4.6.1). The resulting component is always singular.

The components to the left and right of a comparison operator must both be of numeric type, and must both be singular. A compiler error is generated if either is not.

A comparison operator returns TRUE if the magnitude relationship expressed by the operator corresponds to the magnitude relationship of the components to the left and right. Otherwise FALSE is returned.

### 3.7.3.4 Arithmetic Operators

The UIL arithmetic operators are as follows:

| | |
|---|---|
| + | addition |
| - | subtraction |
| * | multiplication |
| / | division |
| MOD | modulo |
| ** | exponentiation |

The arithmetic operators are binary operators that create a component of <numeric_combo> type (Section 3.7.4.6.2). The resulting component has a size (arrayness) equal to the larger of the components to the left and right of the operator.

The components to the left and right of an arithmetic operator must both be of numeric type, and a compiler error is generated if either is not.

The components to the left and right of the operator must be compatible in size. If the size is equal the resulting component will be the appropriate arithmetic combination of the corresponding elements of the left and right components. If one component is singular and the other has more than one element, the resulting plural component is the appropriate arithmetic combination of each element of the plural component with the singular component. If both components are plural, but their size is not equal, an error message is generated by the compiler.

Note that the plus and minus operators may be used in other ways than as simple binary operators. A plus or minus that has no component to its immediate left is interpreted as a sign modifying the component to its right. The resulting component types in these cases are <numeric_pos> and <numeric_neg>. Furthermore, a plus or minus operator preceded, without intervening blanks, by the character "E" and anteceded, without intervening blanks, by any numeric character from 0 to 9, is interpreted as the sign of an exponent within a numeric literal expressed in scientific format.

### 3.7.3.5 List Operators

The UIL "list" operator is the comma.

A list is formed by a series of components separated by commas. The components making up the list may be of any size, and the size of the resulting list component is the sum of the sizes of the constituent components.

If all the components within the list are of boolean type, the resulting component is of the type <boolean_list>. If all the components within the list are of numeric type, the resulting component is of the type <numeric_list>. If all the components within the list are of character string type, the resulting component is of the type <cstring_list>.

If the components within the list are of disparate type, the resulting component is of the type <mixed_list>. However, there is no context in which the <mixed_list> component is legal, and therefore a compiler error will be generated.

Note that a special case of a numeric list component is the "subscript list". A subscript list, enclosed in parentheses, may be used to designate a particular element(s) of an RODB attribute (see Section 3.7.4.2). A subscript list is recognized by its context. A subscript list must consist of components of singular numeric type, or of <wild_card> type (expressed as "*"), or of <range_fixed> type.

### 3.7.3.6 Range Operators

The UIL range operators are as follows:

| | |
|---|---|
| .. | "to" (creates a range) |
| IN | value to the left inside the range on the right |
| OUTSIDE | value to the left outside the range on the right |

The range operator ".." is used to create a range component. The component is of type <range_fixed> if the components on both sides of the operator are numeric literals. Otherwise the resulting component is of type <range_variable>.

The range operators IN and OUTSIDE are used to compare a singular numeric component (on the left) to a range component to the right. The resulting component is of type <boolean_combo>, and is always singular.

A compiler error is generated if the component to the left is not of numeric type, or if it is plural. A compiler error is generated if the component to the right is not a range component.

For the IN operator, the value TRUE is returned if the numeric component to the left is >= the left-hand end of the range and <= the right-hand end of the range. Otherwise FALSE is returned. For the OUTSIDE operator the result is opposite.

Here is an example of a UIL statement that uses range operators:

```
WHEN  SPEED OF PUMP1  IN  300..400
```

The WHEN condition passes if PUMP1's speed is in the range 300 to 400.

### 3.7.3.7 RODB Operators

UIL requires operators that permit the formulation of pairs required in invoking RODB services such as object-attribute read and write, and object-action write. Therefore the following operators are defined:

| | |
|---|---|
| OF | creates an object-attribute pair for reading or writing |
| => | creates a parameter-value pair for use in writing an action |

### 3.7.4 Components

Throughout this Specification reference has been made to "components".

As used in this document the word "component" is simply a general way of referring to certain syntactical elements that recur in various UIL statements.

Components have several attributes: type, size, and variety.

### Component Type

Most components fall into one of three major types. These types are boolean, numeric, and character string.

Boolean components are components that can be evaluated to obtain one or more booleans. Numeric components are components that can be evaluated to obtain one or more "numbers". Character components are components that can be evaluated to obtain a character string.

There are a few components, such as ranges, that do not fall neatly into the three major types. These are discussed separately.

### Component Size

A component's "size" is the number of elements formed by the evaluated component. When they are evaluated all components are considered to be one-dimensional arrays with one or more elements. Thus "size" is equivalent to the "arrayness" of the component.

In this specification the components <singular_boolean> and <singular_numeric> are sometimes referred to. What is meant by such a reference is a component of the boolean type, or numeric type, that has only one element -- an arrayness of one.

### Component Variety

The "variety" of a component may be one of the following:

literal — A fixed value of the appropriate type that is established at compile time. Must not be the object of a SET statement.

object_attribute — An attribute of an object in the SSF RODB available by means of DMS standard services. May be used in decision making or written (SET).

variable — An internal variable created by a DECLARE statement (Section 3.6.1).

constant — A built-in constant of numeric type. Must not be the object of a SET statement.

combination — A value created by combining two other components using an operator of appropriate type. Must not be the object of a SET statement.

definition — A value of any component type that has been given its own name by means of a DEFINE statement (Section 3.6.2). Allowed usage depends upon underlying component type.

list — A series of values created by writing some number of components separated by commas. Must not be the object of a SET statement.

range — A two-valued component formed by linking two numeric components by means of the range operator "..". Must not be the object of a SET statement.

The following table summarizes the component varieties available in UIL:

| VARIETY | BOOLEAN | NUMERIC | CHARACTER | OTHER |
|---|---|---|---|---|
| LITERAL | <boolean_true><br><boolean_false> | <num_scal_lit><br><num_ntgr_lit><br><num_time_lit> | <cstring_lit> | |
| OBJECT-ATTRIBUTE | <bool_obj_attrib> | <num_obj_attrib> | <cstr_obj_attrib> | |
| VARIABLE | <boolean_var> | <numeric_var> | <cstring_var> | |
| CONSTANT | | <numeric_const> | | |
| FUNCTION | | <numeric_funct> | | |
| COMBINATION | <boolean_combo><br><boolean_not> | <numeric_combo><br><numeric_neg><br><numeric_pos> | | |
| DEFINITION | <boolean_def> | <numeric_def> | <cstring_def> | <range_def> |
| LIST | <boolean_list><br><bool_data_list> | <numeric_list><br><num_data_list><br><subscript_list> | <cstring_list><br><cstr_data_list> | <mixed_list> |
| RANGE | | <range_fixed><br><range_variable><br><wild_card> | | |

The component varieties summarized in this table are described in the following sections.

### 3.7.4.1 Literal Components

Literal components may be of boolean, numeric, or character type, as described in the following subsections.

#### 3.7.4.1.1 Boolean Literals

The boolean literals supported by UIL are as follows:

| | | | |
|---|---|---|---|
| <boolean_true> | TRUE | *or* | ON |
| <boolean_false> | FALSE | *or* | OFF |

### 3.7.4.1.2 Numeric Literals

The numeric literals supported by UIL are any alphanumeric strings that can be converted to an 8-byte floating point number or 4-byte integer by the GET routines provided as part of the Ada package TEXT_IO.

In addition UIL supports several forms that are not accepted by TEXT_IO, as follows:

* Numbers that begin with a decimal point, or with a sign followed immediately by a decimal point.

* Numbers that use the familiar DDD:HH:MM:SS.SS format often used to express a time in terms of days, hours, minutes and seconds. Numbers in the formats HH:MM:SS.SS and MM:SS.SS are also accepted. Note that the various fields are not magnitude checked. Thus the forms 96:00:12.1 and 4:00:00:12.1 are both legal, and they are equivalent. The "seconds" field may or may not contain a decimal point. The days, hours, and minutes fields must not contain a decimal point. A time-format literal is converted to seconds for use.

A numeric literals are filed as <num_ntgr_lit> if it is written in the form of an integer, <num_scal_lit> if it contains a decimal point, or <num_time_lit> if it is expressed in time format. However this distinction is maintained only to allow the literal to be printed in an appropriate style during execution. UIL stores all numeric literals as 8-byte floating point numbers.

Here are some examples of valid numeric literals (the right hand column indicates how the number is stored internally):

| | |
|---|---|
| 1000 | 1.00000000000000E+03 |
| -.75 | -7.50000000000000E-01 |
| 1.23456789012345E+67 | 1.23456789012345E+67 |
| 11:11 | 6.71000000000000E+02 |
| 364:23:59:59.999 | 3.15359999990000E+07 |
| 1_222_333_444 | 1.22233344400000E+09 |
| 2#11111111# | 2.55000000000000E+02 |
| 16#AA# | 1.70000000000000E+02 |
| -9999999999999995.0 | -1.00000000000000E+16 |

A numeric literal always has a component size of one. A series of numeric literals separated by commas is interpreted as a list (Section 3.7.3.8) of numeric literals.

### 3.7.4.1.3 Character Literals

A character string literal is any series of characters enclosed by single or double quotation marks. A character string literal may contain blanks. It may not contain single or double quotation marks.

### 3.7.4.2 Object-Attribute Components

UIL recognizes an object-attribute pair by means of the RODB operator "OF". In other words, any component of the form

```
<word_1> OF <word_2>
```

is interpreted as an RODB object-attribute pair. The word preceding the "OF" is treated as an attribute, and the word following the "OF" is treated as an object. Thus any RODB object-attribute pair is of the form

```
<attribute> OF <object>
```

The type of the particular object-attribute component depends on the Ada type of the attribute. Depending on the attribute's type, an RODB object-attribute pair is filed as one of the following components: <bool_obj_attrib>, <num_obj_attrib>, or <cstr_obj_attrib>.

Arrayed attributes of three or fewer dimensions can be referenced in UIL. Particular element(s) of an arrayed attribute are designated by means of a subscript list component enclosed by parentheses. A subscript list consists of an appropriate number of <singular_numeric>, <range_fixed>, or <wild_card> components. If more than one subscript is required, they are separated by commas.

### 3.7.4.3 Internal Variable Components

Variables needed for computations and intermediate values within a bundle may be created by means of the DECLARE statement (Section 3.6.1).

Internal variables may be singular, or may be one-dimensional arrays. The subscripts of an arrayed internal variable always begin with one. In references to arrayed internal variables, subscripts may be expressed as a singular numeric component, a fixed range component, or a "wild card" component (Section 3.7.4.9).

Depending on type, internal variables are filed as one of the following components: <bool_int_var>, <num_int_var>, or <cstr_int_var>.

### 3.7.4.4 Constant Components

UIL contains a number of built-in constants which are available for use in converting from one system of units to another. The constants supported by UIL are the following:

| | |
|---|---|
| DEG_TO_RAD | converts degrees to radians |
| RAD_TO_DEG | converts radians to degrees |
| FT_TO_M | converts feet to meters |
| M_TO_FT | converts meters to feet |
| FT_TO_NM | converts feet to nautical miles |
| NM_TO_FT | converts nautical miles to feet |
| G_TO_FPS2 | converts Gs to feet/sec/sec |
| FPS2_TO_G | converts feet/sec/sec to Gs |

| | |
|---|---|
| LBM_TO_KG | converts pounds mass to kilograms |
| KG_TO_LBM | converts kilograms to pounds mass |
| SLUG_TO_KG | converts slugs to kilograms |
| KG_TO_SLUG | converts kilograms to slugs |
| LBF_TO_N | converts pounds force to newtons |
| N_TO_LBF | converts newtons to pounds force |

Only constants of numeric type are defined, and these are filed as a component of the type <num_constant>. All such constants are singular, i.e., their size (arrayness) is one.

### 3.7.4.5 Function Components

UIL contains a number of built-in functions. The functions supported by UIL are the following:

ABS

SQRT

SIN

COS

TAN

ARCSIN

ARCCOS

ARCTAN

Only built-in functions of numeric type are defined, and these are filed as a component of the type <num_function>. All such functions are singular, i.e., their size (arrayness) is one.

### 3.7.4.6 Combination Components

Combination components are components created by combining other components by means of the operators listed above in Section 3.7.2. Combinations may be of boolean or numeric type. No character string combinations are supported.

### 3.7.4.6.1 Boolean Combinations

Boolean combinations are combinations formed by the use of the operators

AND    OR    NOT    =    /=    <    <=    >    >=

The combination formed by the NOT operator is filed as <boolean_not>. It is of the form:

NOT <boolean_component>

Where <boolean_component> may be any singular or plural boolean component. When evaluated the <boolean_not> is a component of the same size as the <boolean_component>, each of whose elements is the opposite of the corresponding element of the <boolean_component>.

The other forms of the component <boolean_combo> are of the form:

```
<component>  <operator>  <component>
```

In the case of the AND operator and the OR operator, the two components must be of boolean type, and if both are plural their size must be equal. The result is a <boolean_combo> component whose size is equal to the size of the plural component(s), if any.

In the case of the "equals" and "not equals" operators, the two components may be of any type, and if both are plural their size must be equal. The result is a <boolean_combo> component of size one. In the case of the "equals" operator the result is TRUE when the two components are equal element by element, or when each element of a plural component equals a singular component. In the case of the "not equals" operator the result is FALSE under the same conditions.

In the case of the magnitude operators, the two components must be of singular numeric type. The result is a <boolean_combo> component of size one, which is TRUE when the stated magnitude relationship between the two numeric components is true.

#### 3.7.4.6.2 Numeric Combinations

Numeric combinations are combinations formed by the use of the addition (plus), subtraction (minus), multiplication, division, modulo, and exponentiation operators written as follows:

```
+      -      *      /      MOD      **
```

The combination formed by the plus and minus operators is filed as <numeric_pos> or <numeric_neg> if it is of the form:

```
<sign>  <numeric_component>
```

Where <numeric_component> may be any singular or plural numeric component. When evaluated the result is a numeric component each of whose elements is the positive or negative of the corresponding element of the <numeric_component>.

The other configurations of the component <numeric_combo> are of the form:

```
<numeric_component>  <operator>  <numeric_component>
```

The two components must be of numeric type, and if both are plural their sizes must be equal. The resulting component is a <numeric_combo> whose size is equal to the size of the plural component(s), if any. Each element of the resulting component is created by performing the stated arithmetic operation on the corresponding elements of the two components, or on each element of the plural component with respect to the singular component.

The following are examples of numeric combinations:

```
(1, 2, 3)  +  (3, 2, 1)    ==>    (4, 4, 4)
(1, 2, 3)  /  2            ==>    (0.5, 1.0, 1.5)
```

### 3.7.4.7 Definition Components

Components may be formed by means of the DEFINE statement described above in Section 3.6.2. A definition component is classified as a <boolean_def>, <numeric_def>, <cstring_def> or <range_def> depending on whether the component stated in the DEFINE statement is of boolean, numeric, character string, or range type. The size of the definition component is equal to the size of the referenced component.

### 3.7.4.8 List Components

A list is formed by a series of singular or plural components separated by commas. It may optionally be enclosed by parentheses.

A list is classed as a <boolean_list> if every member of the list is a component of boolean type. A list is classed as a <numeric_list> if every member of the list is a component of numeric type. A list is classed as a <cstring_list> if every member is a component of character type.

A list appearing as a subscript to a multi-dimensional array (RODB attribute) is classified as a <subscript_list>. Every element of a subscript list must be a <singular_numeric>, a <range_fixed>, or a <wild_card>.

A list is classed as a <mixed_list> if the members are of more than one type. However, the <mixed_list> component has no legal use in UIL so the creation of a mixed list will result in a compiler error message.

The size of a list component is equal to the sum of the sizes of the member components.

### 3.7.4.9 Range Components

A range component is of the form:

```
      <numeric_component>  ..  <numeric_component>
```

The numeric components used to form a range must be of size one. The resulting range component does not possess a size.

A range is filed as a <range_fixed> component if both numeric components are of numeric literal type, or numeric constant type, or of definition type when the defined component is of fixed or constant type. A range is filed as a <range_variable> if either of the numeric components is of any variable numeric type such as an object-attribute pair, a function, or an internal variable.

There is a special case of a range component known as a <wild_card>. A "wild card" is legal only within a subscript. The range implied is the range corresponding to the allowable subscripts for the particular dimension of the array.

## 4.0 USER-INTERFACE CAPABILITIES

This chapter describes the user-interface capabilities supported by UIL that permit users to monitor and to control the operation of UIL bundles.

Interface between the user and the onboard UIL Executor will be accomplished by the use of RODB actions and attributes.

### 4.1 User Commands

This section describes the user-interface capabilities supported by UIL that permit users to control the operation of UIL.

#### 4.1.1 The FREEZE_ALL Command

The function of the FREEZE_ALL command is to stop the execution of all the bundles currently installed in the UIL Executor. The FREEZE_ALL command thus provides a "panic stop" capability.

The FREEZE_ALL command is implemented as an RODB action. FREEZE_ALL requires no parameters.

When the UIL Executor receives the FREEZE_ALL command all active sequences in all installed bundles are placed in an inactive state. Each sequence is allowed to finish its current statement. The status for each sequence that was active becomes STOPPED_BY_COMMAND. The status of sequences that are already inactive is not changed.

Any sequence stopped by FREEZE_ALL may subsequently be reactivated by means of the START, RESUME, or STEP command.

#### 4.1.2 The INSTALL Command

The function of the INSTALL command is to prepare a bundle for execution by bringing it into local memory and assigning an executor task to process it.

The INSTALL command is implemented as an RODB action. INSTALL requires three parameters: (1) the name that is on the BUNDLE header statement of the bundle to be installed, which must be identical to, or convertible to, the name of the MSU file that contains the executable data for the bundle, (2) the delta-time (DT) at which the bundle should be processed, and (3) the Ada priority (PRIO) at which the bundle should be processed.

When the UIL Executor receives the INSTALL command the executable data for the bundle named by parameter (1) above is read from the Mass Storage Unit and installed in the internal buffers for execution. An executor task is assigned to the bundle, and scheduled at the DT and PRIO provided as parameters (2) and (3) above. The statement pointer associated with each sequence within the bundle is set to the first statement in the sequence. All internal variables declared within the bundle are initialized to null values (see Section 3.3.6.1).

When installation is complete, all sequences within the bundle except those whose sequence header statement contains the word INACTIVE (see Section 3.3.2) are placed in an active state. Other sequences may be activated by means of the START or STEP command.

The INSTALL command is illegal if the named bundle is not currently available as an executable file stored on the Mass Storage Unit. In this case no action occurs and an advisory message is issued.

### 4.1.3 The REMOVE Command

The function of the REMOVE command is to remove an installed bundle from local memory. The bundle remains on the Mass Storage Unit and may be installed again.

The REMOVE command is implemented as an RODB action. REMOVE requires one parameter: the name that is on the BUNDLE header statement of the bundle to be removed.

When the UIL Executor receives the REMOVE command the pointers associated with the removed bundle are reset in such a way that the internal memory utilized by the bundle becomes available for the installation of another bundle.

The REMOVE command is illegal if the named bundle is not currently installed, or if any sequence within the bundle is in an active state. In this case no action occurs and an advisory message is issued.

### 4.1.4 The HALT Command

The function of the HALT command is to stop the execution of all the sequences within a particular bundle. HALT is equivalent to FREEZE_ALL except that it applies only to the sequences inside a single bundle.

The HALT command is implemented as an RODB action. HALT requires one parameter: the name that is on the BUNDLE header statement of the bundle to be halted.

When the UIL Executor receives the HALT command all active sequences in the named bundle are placed in an inactive state. Each sequence is allowed to finish its current statement. The status for each sequence that was active becomes STOPPED_BY_COMMAND. The status of sequences that are already inactive is not changed.

Any sequence stopped by HALT may subsequently be reactivated by means of the START, RESUME, or STEP command.

The HALT command is illegal if the named bundle is not currently installed. In this case no action occurs and an advisory message is issued.

### 4.1.5 The START Command

The function of the START command is to start a particular sequence contained in some bundle.

The START command is implemented as an RODB action. START requires two parameters: (1) the name that is on the BUNDLE header statement of the bundle containing the sequence to be started, and (2) the name that is on the SEQUENCE header statement of the particular sequence. Both the bundle name and the sequence name must be supplied because sequence names are not necessarily unique across bundles.

When the UIL Executor receives the START command the statement pointer associated with the named sequence is set to the first statement within the sequence -- namely the sequence header statement. The sequence is then placed in an active state.

The START command is illegal if the sequence is already in an active state. In this case no action occurs and an advisory message is issued.

### 4.1.6 The STOP Command

The function of the STOP command is to deactivate a particular sequence contained in some bundle.

The STOP command is implemented as an RODB action. STOP requires two parameters: (1) the name that is on the BUNDLE header statement of the bundle containing the sequence to be stopped, and (2) the name that is on the SEQUENCE header statement of the particular sequence. Both the bundle name and the sequence name must be supplied because sequence names are not necessarily unique across bundles.

When the UIL Executor receives the STOP command the indicated sequence is placed in an inactive state, after completing execution of the current statement. The sequence's statement pointer is not changed. Thus a subsequent RESUME or STEP command will cause the sequence to pick up where it left off.

The STOP command is illegal if the indicated sequence is not currently installed. In this case no action occurs and an advisory message is issued.

### 4.1.7 The RESUME Command

The function of the RESUME command is to reactivate a particular sequence contained in some bundle.

The RESUME command is implemented as an RODB action. RESUME requires two parameters: (1) the name that is on the BUNDLE header statement of the bundle containing the sequence to be resumed, and (2) the name that is on the SEQUENCE header statement of the particular sequence. Both the bundle name and the sequence name must be supplied because sequence names are not necessarily unique across bundles.

When the UIL Executor receives the RESUME command the indicated sequence is placed in an active state. The sequence's statement pointer is not changed. Thus if a RESUME follows a FREEZE_ALL, HALT or STOP command, sequence execution will start at the point where it left off. RESUME may also be used to switch to free-running operation following the JUMP_TO or HOLD_AT commands or following STEP operation.

The RESUME command is illegal if the indicated sequence is not currently installed, or if the sequence is not currently in an inactive state. In this case no action occurs and an advisory message is issued.

### 4.1.8 The STEP Command

The function of the STEP command is to execute a single statement in a particular sequence contained in some bundle.

The STEP command is implemented as an RODB action. STEP requires two parameters: (1) the name that is on the BUNDLE header statement of the bundle containing the sequence to be stepped, and (2) the name that is on the SEQUENCE header statement of the particular sequence. Both the bundle name and the sequence name must be supplied because sequence names are not necessarily unique across bundles.

When the UIL Executor receives the STEP command while the indicated sequence is in an active state, the sequence will be deactivated as though in response to the STOP command. If the sequence is inactive, the next statement that would be encountered during free-running operation is executed, and the sequence is then again placed in an inactive state.

The STEP command is illegal if the indicated sequence is not currently installed. In this case no action occurs and an advisory message is issued.

### 4.1.9 The HOLD_AT Command

The function of the HOLD_AT command is to stop a sequence at a particular statement.

The HOLD_AT command is implemented as an RODB action. HOLD_AT requires three parameters: (1) the name that is on the BUNDLE header statement of the bundle containing the sequence to be held, (2) the name that is on the SEQUENCE header statement of the particular sequence, and (3) the number of the statement at which the sequence should stop.

When the UIL Executor receives the HOLD_AT command an indicator will be set that will cause the sequence to be placed in an inactive state the next time (and only the next time) that it encounters the specified statement. The sequence will stop before executing the specified statement. A subsequent RESUME or STEP command will cause the sequence to execute the statement.

The HOLD_AT command is illegal if the indicated sequence is not currently installed, or if the specified statement does not lie within the sequence. In this case no action occurs and an advisory message is issued.

### 4.1.10 The JUMP_TO Command

The function of the JUMP_TO command is to reset the statement pointer associated with a sequence to a new statement.

The JUMP_TO command is implemented as an RODB action. JUMP_TO requires three parameters: (1) the name that is on the BUNDLE header statement of the bundle containing the sequence to be jumped, (2) the name that is on the SEQUENCE header statement of the particular sequence, and (3) the number of the statement to which the sequence should jump.

When the UIL Executor receives a legal JUMP_TO command the statement pointer associated with the indicated sequence is set to the statement number specified by a parameter. The sequence is not activated. The sequence must be explicitly activated by means of the RESUME or STEP commands.

The JUMP_TO command is illegal if the indicated sequence is not currently installed, or if the specified statement does not lie within the sequence, or if the sequence is in an active state. JUMP_TO is also illegal if the specified statement is inside a construct, i.e., a structure begun by a WHEN, WHENEVER, EVERY or IF statement and closed by an END statement. However, jumping out of a construct is allowed. If an illegal jump is commanded no action occurs and an advisory message is issued.

## 4.2 User Monitoring

This section describes the user-interface capabilities supported by UIL that permit users to monitor the operation of UIL.

User monitoring capabilities are supported by providing, as RODB attribute values, onboard status data for display and telemetry.

The following paragraphs describe the status data that make externally available a full description of the current state of the UIL.

### 4.2.1 Bundle Status

This section describes the information made available to describe the status of bundles that are currently installed.

Number of Bundles

An integer attribute NUMBER_OF_BUNDLES reflects the total number of installed bundles.

Bundle Names

A character string attribute BUNDLE_NAME is provided for each installed bundle.

These are the names that appear in the BUNDLE header statements (Section 3.3.1) that mark the beginning of every bundle. Each name should be identical to, or easily convertible to, the name of the MSU file containing the executable data for the bundle.

Bundle Status

An enumeration (or integer) attribute BUNDLE_STATUS gives the status of each installed bundle.

BUNDLE_STATUS may have one of two values: (1) INACTIVE, if the bundle is installed but none of its constituent sequences is active, and (2) ACTIVE, if the bundle is installed and at least one sequence is in an active state.

Bundle DT

A floating-point attribute BUNDLE_DT gives the delta-time at which each installed bundle is processed.

Bundle Priority

An integer attribute BUNDLE_PRIO gives the Ada priority at which each installed bundle is processed.

Number of Sequences in Bundle

An integer attribute BUNDLE_N_SEQS gives the number of sequences that are present in each installed bundle.

User Information for Bundle

A character string attribute BUNDLE_USER_INFO is reserved for descriptive information for each bundle. This attribute is not set by the UIL executor. It is available for use by the user. This attribute is limited to 32 characters.

### 4.2.2 Sequence Status

This section describes the information that describes the status of the sequences contained in the bundles that are currently installed. The number of sequences in each bundle is provided by the BUNDLE_N_SEQS attribute described above.

Sequence Names

A character string attribute SEQUENCE_NAME gives the name of each sequence.

The names made available in this attribute are the names that appear in the SEQUENCE header statements (Section 3.3.2) that mark the beginning of every sequence.

Sequence Status

An enumeration (or integer) attribute SEQUENCE_STATUS gives the status of each sequence.

SEQUENCE_STATUS may have one of five values: (1) ACTIVE, if the sequence is active, (2) NEVER_STARTED, if the sequence is inactive and has never been active since the installation of the parent bundle, (3) FINISHED, if the sequence is inactive because it has reached an end, (4) STOPPED_BY_COMMAND, if the sequence is inactive because it was stopped by a user command such as STOP, STEP, or HOLD_AT (Section 4.1), and (5)

STOPPED_BY_ERROR, if the sequence is inactive because it was stopped by an error encountered during the processing of any statement.

Sequence Start Time

A floating-point attribute SEQUENCE_START_TIME gives the time at which each sequence was first started.

The SEQUENCE_START_TIME attribute is set to the current GMT time whenever the SEQUENCE header statement (Section 3.3.2) of the sequence is executed.

Sequence Stop Time

A floating-point attribute SEQUENCE_STOP_TIME gives the time at which each sequence ended.

The SEQUENCE_STOP_TIME attribute is set to the current GMT time whenever the CLOSE statement (Section 3.3.4) that marks the end of the sequence is executed.

Sequence Statement

An integer attribute SEQUENCE_STATEMENT gives the number of the statement currently being executed by each sequence.

If the sequence is active the SEQUENCE_STATEMENT attribute gives the number of the statement currently being processed. If the sequence is inactive the SEQUENCE_STATEMENT attribute points to the statement that would be processed first if the sequence were made active by a user command such as RESUME or STEP. Note that UIL statements are numbered from the start of the parent bundle rather than being numbered within each sequence and subsequence.

User Information for Sequence

An arrayed character string attribute SEQUENCE_USER_INFO is reserved for descriptive information for each sequence. This attribute is not set by the UIL executor. It is available for use by the user. This attribute is limited to 32 characters.

### 4.2.3 Text Information

This section describes an attribute that is made available to support the display of text information. This attribute is named UIL_DISPLAY_TEXT and is a character string of size and arrayness to be determined.

Such information may be required to support a "selection list" field in a "UIL Monitor and Control" display that will present a table of contents of the bundles, and their constituent sequences, that are currently installed.

Such a table of contents could be compiled externally to the UIL executor from information made available in the attributes described in Section 4.2. The UIL_DISPLAY_TEXT attribute is defined in case it is necessary for the "table of contents" to be created in text form by the UIL executor.

The UIL_DISPLAY_TEXT attribute may also be used to support an automatically scrolling display of the text of a particular sequence, if such a display becomes a requirement.

## Appendix A

## Abbreviations and Acronyms

# Appendix B

## Glossary

# User's Guide to the TIMELINER Language

## Ada-Language Version

Don Eyles

Concetta Cuevas

March 6, 1991

## TABLE OF CONTENTS

## 1.0 INTRODUCTION

This document describes a language called TIMELINER that provides capabilities allowing a user to initialize, control and debug a simulation by means of a script written ahead of time.

This document serves both as a User's Guide and as the principal descriptive document for the Ada-language version of the TIMELINER language.

### 1.1 Purpose of TIMELINER

TIMELINER was created because of the inadequacy of the software tools that were available 10 years ago at the Draper Laboratory for supplying initialization information to simulations of space systems.

TIMELINER provides the capability both for time-zero initialization and for inputting information during the simulation at times or under conditions that are specified by the user. TIMELINER scripts can be created to play the role of a "paper pilot".

In addition to its function as a flexible initialization tool, the TIMELINER language can be used as a debugging aid. For example, a TIMELINER sequence may be constructed that will assist the capture of information needed to analyze an anomaly.

Versions of TIMELINER now exist in the HAL, Fortran, and Ada languages, as appropriate to the simulations in which they play a role. This document describes the Ada-language version. Appendix A outlines the differences between this and earlier versions of TIMELINER. Users familiar with the HAL or Fortran version may find it useful to refer to this appendix. Others should ignore it.

The TIMELINER language is in many ways analogous to the NASA Space Station User Interface Language (UIL), especially to the "compiled" form of UIL intended for the implementation of "procedures" for automated control of the space station and its systems. Besides its primary purpose as a simulation tool, the Ada-language version of TIMELINER serves as a test-bed for evaluating concepts related to UIL.

Users constitute the best judges of any system such as TIMELINER that is designed to be "easy to use". User comments are therefore solicited, both with respect to the language and to this User's Guide. We will do our best to respond to comments with appropriate changes.

### 1.2 Basic Principles

The central concept of the TIMELINER language is that the user must be allowed to specify an arbitrary number of independent sequences. Within each sequence processing proceeds sequentially from top to bottom as defined by the constructs used. Meanwhile the various sequences operate in parallel with each other.

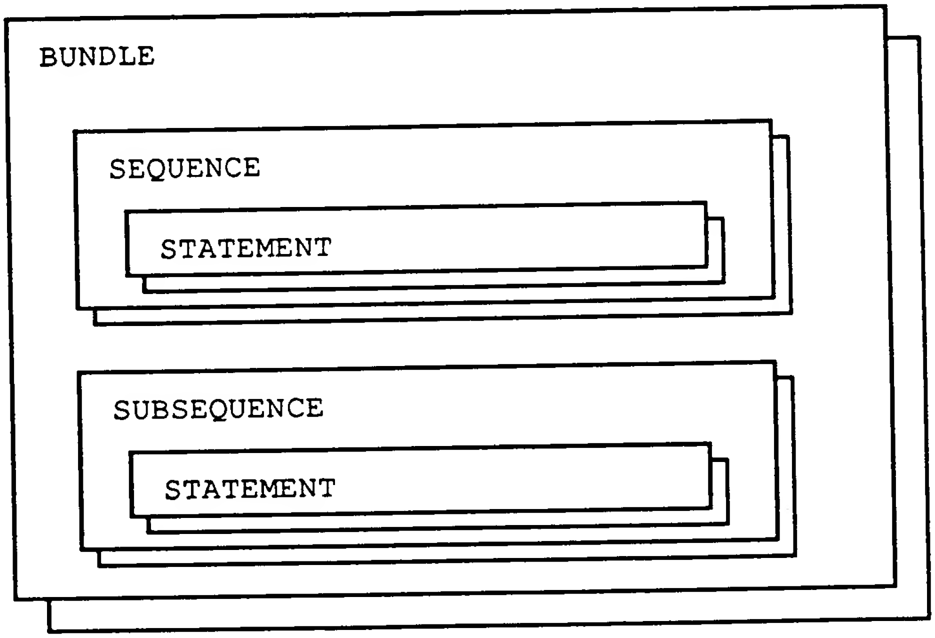
TABLE OF CONTENTS (CONTINUED)

## 2.0 STRUCTURE OF TIMELINER SCRIPTS

This section describes the hierarchical structure of a TIMELINER script. The "blocking statements" that implement this structure are described in the following chapter.

The highest level of the TIMELINER hierarchy is the "bundle" -- used for grouping and packaging the "sequences" and "subsequences" that constitute the substance of a TIMELINER script. Sequences and subsequences, in turn, are made of "statements".

The TIMELINER hierarchy is illustrated by the following picture:



### 2.1 The "Bundle"

The uppermost hierarchical level of a TIMELINER script is the "bundle". The concept of "bundle" is new to the Ada-language version of TIMELINER.

The bundle is defined as a grouping of sequences and subsequences that form a whole, either because of some common purpose, or because they are targeted for use in a particular operating environment. For example, in the context of a multi-processor system such as the Space Station DMS (or the Draper Lab simulation thereof), the bundle may be used to group sequences and subsequences intended for execution in a particular processor.

A bundle may contain any number of sequences and subsequences. However, a subsequence must lie within the same bundle as the CALL statement that invokes it.

Such a "serial/parallel" capability allows input streams to be organized in accordance with whatever principles seem appropriate to the user. The user is freed from the necessity to integrate all functions into a single input stream.

TIMELINER streams may be broken down by discipline (e.g. flight control, guidance, crew input) or by any other rule that is convenient. TIMELINER includes the capability of creating "subsequences" that may be called by another TIMELINER sequence.

A particular timeline sequence may be quite small: for example a single "whenever" construct specifying an action to occur whenever a particular condition occurs. On the other hand a sequence may be quite long: for example a long string of "when" constructs specifying actions to be taken, in turn, as a series of conditions become true.

## 1.3 How to Use TIMELINER

TIMELINER operation has two parts:

*Compile time:*

At "compile time" raw TIMELINER scripts provided by the user are parsed, checked for errors, and tabulated. A compile-time "listing" is provided. If free from errors the script is stored in an ASCII file in a form suitable for execution. The name of the file is determined by the name of the "bundle" that constitutes the uppermost level of the TIMELINER hierarchy.

At compile time TIMELINER operates in "batch" fashion. The TIMELINER compile capability is installed on the MicroVAX computer that serves as the "development host" for the Draper Lab real-time test bed of the Space Station DMS. Compile capability may also be hosted by individual Macintosh computers.

*Run time:*

At "run time" the executable-code file corresponding to a particular TIMELINER script is read from the file and executed. During execution-time TIMELINER is called repetitively, at frequent intervals. The spacing of these intervals determines the "granularity" with which TIMELINER is capable of timing functions such as WAIT.

TIMELINER prints TIMELINER statements when they are executed. This execution-time printing forms part of the text output stream of the simulation that can be displayed on a terminal or printed on paper.

Subsequence calls may be nested. That is, a subsequence may call another subsequence. In every case, however, a subsequence forms part of the stream of execution established by the sequence that makes the upper-level call. A subsequence that is never called will compile -- but of course it cannot be executed.

Note that while subsequence calls may be nested, subsequences themselves appear in a TIMELINER script at the same level as sequences. A subsequence should not be placed physically inside a sequence.

A subsequence must belong to the same bundle as the sequence(s) or subsequence(s) that call it. That is, subsequence calls may not be made across bundle boundaries. An error message (Cuss 58) is issued at TIMELINER compile-time, during processing of the CLOSE BUNDLE statement, if a called subsequence is not found in the bundle.

The statements making up the subsequence have the same effect as they would have if they physically were part of the calling sequence. However, constructs must be fully contained within a subsequence. That is, a WHEN construct (for example) may not be initiated in a sequence and closed in a called subsequence, nor vice versa.

Users typically employ subsequences to encapsulate particular procedures that may be required as part of a sequence that performs upper-level (e.g. mission) sequencing. This simplifies the appearance of the upper-level sequence and removes the particular procedure, which may seldom change, from the upper-level sequence where changes may be more frequent.

A subsequence is initiated by a SUBSEQUENCE header statement (described in Section 3.3 below) and terminated by a CLOSE SUBSEQUENCE statement (Section 3.4).

## 2.4 TIMELINER "Statements"

Bundles contain sequences and subsequences. Sequences and subsequences, in turn, contain statements.

TIMELINER statements are often referred to as "lines" because in fact a statement must be confined to a single line and multiple statements may not be present on the same line. The end of a line is understood by the TIMELINER parser as ending the statement in question.

The length available for each input line is currently set at 132 characters.

One exception to the identity between statements and lines exists in TIMELINER. A LOAD statement (described in Section 5.1 below) may require more initialization data than will fit conveniently on one line. Therefore LOAD statements are permitted to use multiple lines.

The semi-colon is not required to end a TIMELINER statement. However, to avoid aggravating Ada-hounds for whom ending statements with a semi-colon has become second nature, semi-colons are permitted.

TIMELINER statements are of three types: "blocking" statements, "control" statements, and "action" statements:

A TIMELINER script may contain any number of bundles. A separate executable-code file is produced for each bundle. The name of the file created for each bundle is

```
      TL_<name_of_bundle>.DATA
```

A bundle is initiated by a BUNDLE header statement (described in Section 3.1 below) and terminated by a CLOSE BUNDLE statement (Section 3.4).

Note that all TIMELINER statements, except comments, must lie inside a bundle. Therefore the first statement in any TIMELINER script must be a BUNDLE header statement and the last must be a CLOSE BUNDLE statement.

## 2.2 The "Sequence"

A given TIMELINER bundle contains one or more "sequences" and "subsequences". Sequences and subsequences in turn contain TIMELINER statements.

The essence of a sequence is to establish an independent stream of execution. That is, a sequence is processed <u>in parallel</u> with other sequences. The TIMELINER user may organize a script into any number of sequences as required to accomplish his purpose.

Parallelism among TIMELINER sequences offers several advantages. When two or more conditions requiring a response may occur in an order that is unknown ahead of time -- a situation that would be very difficult to handle within a single sequential stream -- a TIMELINER user may set up multiple sequences to deal with the disparate situations. A new function does not have to be integrated into an existing stream of execution. It may be implemented as an additional sequence.

A TIMELINER script may be as short as a single WHENEVER construct that performs an action <u>whenever</u> a specified condition occurs, or as long as a string of WHEN constructs and WAIT statements that provides the master sequencing for the simulation of an entire mission. Many TIMELINER scripts tend to be organized into sequences of these two types.

A sequence is initiated by a SEQUENCE header statement (described in Section 3.2 below) and terminated by a CLOSE SEQUENCE statement (Section 3.4).

Note that, aside from comments, all TIMELINER statements except those that initiate and conclude a bundle must lie within some sequence or subsequence. Therefore the second statement in any TIMELINER script must be a sequence (or subsequence) header statement.

## 2.3 The "Subsequence"

A TIMELINER "subsequence" is in many ways similar to a sequence. The difference is that while a sequence always forms a new stream of execution, a subsequence is executed within the stream of execution created by the sequence that calls it by means of a CALL statement (Section 4.11).

| | |
|---|---|
| AND | condition conjunction (6.1) |
| OR | condition conjunction (6.1) |
| XOR | condition conjunction (6.1) |
| ON | boolean literal (6.4.1) |
| OFF | boolean literal (6.4.1) |
| TRUE | boolean literal (6.4.1) |
| FALSE | boolean literal (6.4.1) |
| MOD | arithmetic operator (6.4.7) |
| OF | function indicator (6.4.7) |
| ABS | arithmetic function (6.4.7) |
| SQRT | arithmetic function (6.4.7) |
| SIN | arithmetic function (6.4.7) |
| SINE | arithmetic function (6.4.7) |
| COS | arithmetic function (6.4.7) |
| COSINE | arithmetic function (6.4.7) |
| TAN | arithmetic function (6.4.7) |
| TANGENT | arithmetic function (6.4.7) |
| ARCSIN | arithmetic function (6.4.7) |
| ARCSINE | arithmetic function (6.4.7) |
| ARCCOS | arithmetic function (6.4.7) |
| ARCCOSINE | arithmetic function (6.4.7) |
| ARCTAN | arithmetic function (6.4.7) |
| ARCTANGENT | arithmetic function (6.4.7) |

Since TIMELINER determines the meaning of a word in relation to the context in which it appears, these words may in some cases be available to users. For example, a sequence could be named ARCTANGENT without creating a problem. However, as a general practice users should avoid chosing any of the words listed above as names of bundles, sequences, subsequences, or variables.

Additional reserved words of the form xx_TO_xx may exist depending on what scale factors are implemented as part of the scale factor capability of the LOAD statement described below in Section 5.1.

## 2.6 Comments and Blank Lines

TIMELINER permits comments to be added to a script. Such comments are visible in the raw script and in the compile-time listing of the script. As in any computer language, comments are not retained in the executable code.

The Ada-language version of TIMELINER adopts the Ada language's convention in regard to comment designation. Two or more adjacent hyphens (i.e. minus signs) mark the beginning of a comment. All material between the first double hyphen on a line and the end of the line is considered to be part of the comment. A comment may occupy a line by itself, or share a line with a TIMELINER statement.

Blank lines are legal and the spacing implied by such blank lines is retained in the compile-time listing of the script.

Blocking statements are those used to initiate and conclude TIMELINER bundles, sequences, and subsequences. These statements are described in Chapter 3 of this User's Guide.

Control statements are those associated with the WHEN, WHENEVER, EVERY and IF constructs that are used to specify the conditions for TIMELINER actions. The WAIT and CALL functions are also considered to be "control" statements. Control statements are described in Chapter 4.

Action statements stand alone in themselves and perform some "action" such as loading or printing a variable. LOAD and PRINT, for example, are action statements. Action statements are described in Chapter 5.

## 2.5 TIMELINER "Words"

Statements are made up of words. A TIMELINER "word" is defined as a string of characters that lie between blanks. A blank is allowed within a "word" only in the case of a <string_literal> bounded by single or double quotation marks.

Note that arithmetic, comparative, and logical operators must be separated by blanks from the words on either side. Thus the following will not parse properly (the correct form is shown on the right):

```
TIME/10                        TIME / 10
S1="ATTITUDE HOLD"             S1 = "ATTITUDE HOLD"
TIME < TIG&F1 >= F2            TIME < TIG & F1 >= F2
```

Certain words are "reserved" for special purposes in the TIMELINER language. TIMELINER's reserved words include the following:

| | |
|---|---|
| BUNDLE | blocking statement type (3.1) |
| SEQ | blocking statement type (3.2) |
| SEQUENCE | blocking statement type (3.2) |
| SUB | blocking statement type (3.3) |
| SUBSEQ | blocking statement type (3.3) |
| SUBSEQUENCE | blocking statement type (3.3) |
| CLOSE | blocking statement type (3.4) |
| WHEN | control statement/construct type (4.1) |
| WHENEVER | control statement/construct type (4.2) |
| EVERY | control statement/construct type (4.3) |
| IF | control statement/construct type (4.4) |
| BEFORE | control statement type (4.5) |
| WITHIN | control statement type (4.6) |
| OTHERWISE | control statement type (4.7) |
| ELSE | control statement type (4.8) |
| END | control statement type (4.9) |
| WAIT | control statement type (4.10) |
| CALL | control statement type (4.11) |
| LOAD | action statement type (5.1) |
| ALL | LOAD statement modifier (5.1.1) |
| FROM | LOAD statement modifier (5.1.2) |
| PRINT | action statement type (5.2) |
| ABORT | action statement type (5.3) |
| ACTION | action statement type (5.4) |

## 3.3 The SUBSEQUENCE Header Statement

Subsequences are like sequences except that they form part of the stream of execution of the sequence that calls the subsequence. The role of the "subsequence" is discussed in the previous chapter.

A subsequence is initiated by a header statement of the form:

```
SUB[SEQ[UENCE]]  <name_of_subsequence>
```

where <name_of_subsequence> is an alphanumeric word without imbedded blanks chosen by the user. In the present implementation such names are limited to 32 characters.

Each subsequence is closed by a CLOSE SUBSEQUENCE statement, as described below in Section 3.4.

## 3.4 The CLOSE Statement

The CLOSE statement is used to conclude TIMELINER bundles, sequences, and subsequences.

A bundle is ended by the statement

```
CLOSE  BUNDLE  [<name_of_bundle>]
```

When compile-time processing reaches a CLOSE BUNDLE statement the file containing the executable code for the bundle is closed. A new file will be opened if the script contains any additional bundles. This statement has no function during execution-time.

A sequence is ended by the statement

```
CLOSE  SEQ[UENCE]  [<name_of_sequence>]
```

When execution-time processing reaches a CLOSE SEQUENCE statement the sequence in question is concluded and placed in an inactive state.

A subsequence is ended by the statement

```
CLOSE  SUB[SEQ[UENCE]]  [<name_of_subsequence>]
```

When execution-time processing reaches a CLOSE SUBSEQUENCE statement processing returns to the sequence or subsequence containing the CALL statement that invoked the subsequence, at the statement immediately following the CALL statement.

In all three types of the CLOSE statement, the user may optionally include the name of the block being closed as the third word in the statement. In some cases providing this information may make a script easier to read. If the block name is included, TIMELINER will issue an error message (Cuss 26) if the name that is given does not match the name of the innermost open block.

## 3.0 BLOCKING STATEMENTS

This chapter describes the "blocking" statements that are used to initiate and conclude TIMELINER bundles, sequences, and subsequences.

### 3.1 The BUNDLE Header Statement

The uppermost hierarchical level of the Ada-language version of TIMELINER is the "bundle". The role of the "bundle" is discussed in the previous chapter.

A bundle is initiated by a header statement of the form:

```
BUNDLE <name_of_bundle>
```

where <name_of_bundle> is an alphanumeric word without imbedded blanks chosen by the user. Underscores are allowed. In the present implementation such names are limited to 32 characters.

A bundle header statement is never "executed", as such. However, the bundle name determines the name of the file to which the compile-time processing outputs the executable form of the script. The name of the file created for each bundle is

```
TL_<name_of_bundle>.DATA
```

If a raw script contains multiple bundles then multiple executable files are created.

Each bundle is closed by a CLOSE BUNDLE statement, as described below in Section 3.4.

### 3.2 The SEQUENCE Header Statement

Multiple sequences, which execute in parallel with each other, contain the substance of a TIMELINER script. The role of the "sequence" is discussed in the previous chapter.

A sequence is initiated by a header statement of the form:

```
SEQ[UENCE]  <name_of_sequence>
```

where <name_of_sequence> is an alphanumeric word without imbedded blanks chosen by the user. In the present implementation such names are limited to 32 characters.

Each sequence is closed by a CLOSE SEQUENCE statement, as described below in Section 3.4.

## 4.0 CONTROL STATEMENTS

This chapter describes the "control" statements provided to permit the TIMELINER user to specify the conditions under which actions are to occur.

Some TIMELINER control statements initiate "constructs" -- namely the WHEN, WHENEVER, EVERY and IF statements. Such constructs are concluded by an END statement. Other control statements, such as BEFORE, WITHIN, OTHERWISE, ELSE, and ELSE IF, are used within these constructs. Two additional, stand-alone statements are classified as control statements, namely WAIT and CALL.

### 4.1 The WHEN Construct

The WHEN statement is used to specify conditions at which an action is to occur -- on a one-shot basis. The WHEN statement initiates a WHEN construct.

A WHEN construct in its simplest form consists of a WHEN statement, an END statement, and the statements that lie between. The simple WHEN construct is described in Section 4.1.1 below.

A WHEN construct may optionally be modified by a BEFORE or WITHIN statement, and if so modified it may include an OTHERWISE statement. The modified WHEN construct is described in Section 4.1.2 below.

### 4.1.1 The Simple WHEN Construct

The simple form of the WHEN construct is as follows:

```
WHEN  <condition>
      <statements>
END  [WHEN]
```

The WHEN statement, when encountered, causes the sequence of which it is a part to pause until the condition that forms part of the statement is fulfilled. When the condition is met, the statements following the WHEN statement are processed until the END statement is encountered. Processing then "drops through" to the statements following the END statement.

The syntactical element <condition> is defined and discussed in Section 6.1 below.

The statements within a WHEN construct may be any action or control statement, including nested control constructs.

WHEN statements are most useful in the case of a TIMELINER sequence being used to specify a long string of actions. Each WHEN construct may be thought of as a gate across a straight road. The stated condition must occur in order for the gate to allow the traveller to proceed.

### 3.5 Example of TIMELINER Blocking

The following example illustrates the upper level organization of a TIMELINER script:

```
      BUNDLE MISSION_SEQUENCING

            SEQUENCE MAIN
                  <statement>
                  CALL LOADER
                  <statement>
            CLOSE SEQUENCE MAIN

            SEQUENCE SECONDARY
                  <statement>
                  <statement>
            CLOSE SEQUENCE SECONDARY

            SUBSEQUENCE LOADER
                  <statement>
                  <statement>
            CLOSE SUBSEQUENCE LOADER

      CLOSE BUNDLE MISSION_SEQUENCING
```

The executable version of this script would be placed in a file of the name:

```
      TL_MISSION_SEQUENCING.DATA
```

A modified WHEN construct may also be thought of as executing separate statements depending upon which of two conditions occurs first, such as in the English statement:

> When the pot boils before the timer buzzes, turn off the heat, otherwise reset the timer.

or the statement:

> When the pot boils within 10 minutes, turn off the heat, otherwise turn up the heat.

The syntactical element <condition> used by the BEFORE statement is defined and discussed in Section 6.1 below. The syntactical element <time_interval> used by the WITHIN statement is defined and discussed in Section 6.3 below.

The statements within the WHEN construct, including those in the OTHERWISE clause, may be any action or control statements, including nested control constructs.

To return to the metaphor of the WHEN statement as a gate across a straight road, fulfillment of a BEFORE or WITHIN condition creates a path that allows the traveller to bypass the obstacle if the specified condition is met. The OTHERWISE clause, if present, lies along the bypass path.

Here is an example of the modified WHEN construct:

```
WHEN ACCEL < 0.3
   WITHIN 10
      <statement>
      <statement>
END WHEN
```

In this example the sequence waits until the acceleration given by the variable ACCEL is less than 0.3, or until 10 seconds have elapsed since that wait began. If the time interval elapses before or at the same time as the WHEN condition is met, then processing skips to the END statement (and to the statements that follow) without executing the inside statements. If the ACCEL condition is met first, the inside statements are executed before reaching the END statement.

Here is another example of the modified WHEN construct, this time including an OTHERWISE clause:

```
WHEN VIEW_ANGLE > 15
   BEFORE LENS_TEMP > 212 or SENSOR_VOLTS > 25.4
      <statement>
      <statement>
   OTHERWISE
      <statement>
      <statement>
END WHEN
```

In this example two conditions are waited for, the WHEN condition expressed in terms of VIEW_ANGLE, and the BEFORE condition involving lens temperature and sensor voltage. If the WHEN condition occurs first the first set of inside statements is executed. Otherwise the second

Here is an example of the use of the simple WHEN construct:

```
WHEN ACCEL < 0.1
      <statement>
      <statement>
END WHEN
WHEN TIME >= 11:11:11 OR ABORT_FLAG = TRUE
      <statement>
      <statement>
END WHEN
```

In this case the sequence waits until the acceleration denoted by the variable ACCEL is less than 0.1, then executes the statements within the construct. The sequence then waits until either time reaches 11 hours, 11 minutes, and 11 seconds, or the boolean ABORT_FLAG becomes true; and then executes the statements within the second WHEN construct. The sequence then continues at the statements that follow.

### 4.1.2 The Modified WHEN Construct

More complicated forms of the WHEN construct may be created by use of the BEFORE or WITHIN statement. Furthermore, if WHEN is modified by BEFORE or WITHIN, an OTHERWISE statement may be used.

The modified form of the WHEN construct is as follows:

```
WHEN  <condition>
   [BEFORE  <condition>  |  WITHIN  <time_interval>]
      <statements>
   [OTHERWISE
      <statements>]
END  [WHEN]
```

The WHEN statement, when encountered, causes the sequence of which it is a part to pause until the condition that forms part of the WHEN statement is fulfilled. When the condition is met, the statements following the BEFORE or WITHIN statement are processed until either an OTHERWISE or the END statement is encountered. Processing then continues at the statements following the END statement.

However, each time the WHEN condition is to be checked, the condition specified by the BEFORE or WITHIN statement is evaluated. Regardless of whether the WHEN condition is fulfilled simultaneously, if the condition that forms part of the BEFORE statement is fulfilled, or if the time interval specified in the WITHIN statement has elapsed since the WHEN statement was encountered, then processing continues at the statements following the OTHERWISE statement. If there is no OTHERWISE statement, processing skips straight to the END statement and continues at the statements following the END.

In other words, the BEFORE and WITHIN statement may be used to terminate operation of the WHEN construct if a condition (BEFORE) is met, or if a time interval (WITHIN) elapses. An OTHERWISE statement allows a separate set of statements to be provided that will be processed when the construct is terminated.

failure occurs. If, following processing of the statements inside the construct, the jet failure indication is still present, the statements will not be processed again immediately. They will be processed again if the jet failure is reset and subsequently reappears.

(Note that if the flag is turned off and on during the time that the interior statements are being processed, these transitions will not be sensed by the WHENEVER construct. The construct will act as though the condition were continuously present.)

### 4.2.2 The Modified WHENEVER Construct

More complicated forms of the WHENEVER construct may be created by use of the BEFORE or WITHIN statement..

The modified form of the WHENEVER construct is as follows:

```
WHENEVER  <condition>
   [BEFORE  <condition>  |  WITHIN  <time_interval>]
      <statements>
END  [WHENEVER]
```

The WHENEVER statement, when encountered, causes the sequence of which it is a part to pause until the condition that forms part of the WHENEVER statement is fulfilled. When the condition is met, the statements following the BEFORE or WITHIN statement are processed until the END statement is encountered. Then processing returns to the WHENEVER statement. The loop repeats upon the next off-to-on transition of the WHENEVER condition.

However, each time the WHENEVER condition is to be checked, the condition established by the BEFORE or WITHIN statement is evaluated. Regardless of whether the WHENEVER condition is fulfilled, if the condition that forms part of the BEFORE statement is fulfilled, or if the time interval specified in the WITHIN statement has elapsed since the WHENEVER statement was first encountered, then processing skips straight to the END statement and drops through to the statements that follow the END statement.

In other words, the BEFORE and WITHIN statement may be used to terminate the loop created by the WHENEVER statement if a (BEFORE) condition is met, or if a (WITHIN) time interval elapses. Without a BEFORE or WITHIN statement a WHENEVER construct will loop indefinitely.

The syntactical element <condition> used by the BEFORE statement is defined and discussed in Section 6.1 below. The syntactical element <time interval> used by the WITHIN statement is defined and discussed in Section 6.3 below.

The modified WHENEVER construct is useful when a TIMELINER sequence needs to perform certain actions whenever a condition is fulfilled, up until some other condition occurs or until a time interval elapses. Returning to the analogy of a gate across a circular road, a BEFORE or WITHIN clause establishes a path out of the circle that will be taken upon fulfillment of the BEFORE or WITHIN condition.

set is executed. In either case processing then proceeds to the END statement and to the statements that follow it.

### 4.2 The WHENEVER Construct

The WHENEVER statement is used to specify conditions at which an action is to occur -- on a repeating basis. The WHENEVER statement initiates a WHENEVER construct.

A WHENEVER construct in its simplest form consists of a WHENEVER statement, an END statement, and the statements that lie between. The simple WHENEVER construct is described below in Section 4.2.1.

A WHENEVER construct may optionally be modified by a BEFORE or WITHIN statement. The modified WHENEVER construct is described below in Section 4.2.2.

#### 4.2.1 The Simple WHENEVER Construct

The simple form of the WHENEVER construct is as follows:

```
WHENEVER  <condition>
      <statements>
END  [WHENEVER]
```

The WHENEVER statement, when encountered, causes the sequence of which it is a part to pause until the condition that forms part of the statement is fulfilled. When the condition is met, the statements following the WHENEVER statement are processed until the END statement is encountered. Then processing returns to the WHENEVER statement. Upon the next off-to-on transition of the condition stated in the WHENEVER statement the interior statements are again processed. This loop repeats indefinitely.

The syntactical element <condition> is defined and discussed in Section 6.1 below.

The statements within a WHENEVER construct may be any action or control statements, including nested control constructs.

WHENEVER constructs are most useful when a TIMELINER sequence is being used to perform certain actions <u>whenever</u> a particular condition is fulfilled. A WHENEVER construct may be thought of as a gate across a circular road. The stated condition must occur in order for the gate to open, but the traveller will encounter the same gate again and must again wait for the condition to become true.

Here is an example of the use of the simple WHENEVER construct:

```
WHENEVER JET_FAIL = TRUE
      <statement>
      <statement>
END WHENEVER
```

In this example, when the sequence reaches the WHENEVER construct a loop is started that will process two statements at any time that a jet

The syntactical element <time_interval> is defined and discussed in Section 6.3 below.

The statements within an EVERY construct may be any action or control statements, including nested control constructs.

An EVERY constructs is most useful when a TIMELINER sequence is being used to perform certain actions at intervals. An EVERY construct may be thought of as a gate across a circular road which opens at intervals.

Here is an example of the use of the simple WHENEVER construct:

```
EVERY DT_NAV
      <statement>
      <statement>
END EVERY
```

In this example, when the sequence reaches the EVERY construct a loop is started that will process two statements at intervals corresponding to the navigation delta-time given by the variable DT_NAV. The loop continues indefinitely.

### 4.3.2 The Modified EVERY Construct

More complicated forms of the EVERY construct may be created by use of the BEFORE or WITHIN statement.

The modified form of the EVERY construct is as follows:

```
EVERY  <time_interval>
   [BEFORE  <condition>  |  WITHIN  <time_interval>]
      <statements>
END  [EVERY]
```

The EVERY statement, when first encountered, immediately drops through to execute the statements that lie between the BEFORE or WITHIN statement and the END statement. When the END statement is reached, processing returns to the EVERY statement. When the time interval specified by the EVERY statement has elapsed, the interior statements are again processed.

However, on each pass the condition established by the BEFORE or WITHIN statement is evaluated. If the condition that forms part of the BEFORE statement is fulfilled, or if the time interval specified in the WITHIN statement has elapsed since the EVERY statement was first encountered, then processing skips straight to the END statement and drops through to the statements that follow the END statement.

In other words, the BEFORE and WITHIN statement may be used to terminate the loop created by the EVERY statement if a (BEFORE) condition is met, or if a (WITHIN) time interval elapses. Without a BEFORE or WITHIN statement an EVERY construct will loop indefinitely.

The syntactical element <condition> used by the BEFORE statement is defined and discussed in Section 6.1 below. The syntactical element <time interval> used by the WITHIN statement is defined and discussed in Section 6.3 below.

Here is an example of the modified WHENEVER construct:

```
WHENEVER DAP_MODE = "AUTO"
   BEFORE MAJOR_MODE /= 301
      <statement>
      <statement>
END WHENEVER
WHENEVER DAP_MODE = "AUTO"
   BEFORE MAJOR_MODE /= 302
      <statement>
      <statement>
END WHENEVER
```

In this case the sequence processes two statements whenever digital autopilot mode (a character string) goes to automatic, for as long as MAJOR_MODE is 301. Execution then moves to the second WHENEVER construct, which executes different statements whenever autopilot mode goes to automatic during major mode 302.

### 4.3 The EVERY Construct

The EVERY statement is used to make some actions occur repetitively at some specified time interval. The EVERY statement initiates an EVERY construct.

An EVERY construct in its simplest form consists of an EVERY statement, an END statement, and the statements that lie between. The simple EVERY construct is described below in Section 4.3.1.

An EVERY construct may optionally be modified by a BEFORE or WITHIN statement. The modified EVERY construct is described below in Section 4.3.2.

#### 4.3.1 The Simple EVERY Construct

The simple form of the EVERY construct is as follows:

```
EVERY  <time_interval>
      <statements>
END  [EVERY]
```

The EVERY statement, when first encountered, immediately drops through to execute the statements that lie between the EVERY statement and the END statement. When the END statement is reached, processing returns to the EVERY statement. When the time interval specified by the EVERY statement has elapsed, since the previous encounter with the EVERY statement, the interior statements are again processed. This loop repeats indefinitely.

The time interval in an EVERY statement may be specified as a literal or a variable. If the time interval is specified as a variable, the variable is re-evaluated each time the EVERY statement is processed.

The form of the IF construct is as follows:

```
IF  <condition>
     <statements>
{ELSE  IF  <condition>
     <statements>}
[ELSE
     <statements>]
END  [IF]
```

When the IF statement is encountered, its condition is evaluated. If the condition passes, the statements that follow the IF statement are processed, and execution then skips to the END statement and falls through to whatever statements follow. If the condition fails, and there is no ELSE IF or ELSE clause, execution skips straight to the END statement.

If there are any ELSE IF or ELSE clauses, they are processed in the obvious way. That is, if the IF condition fails, execution skips to the first ELSE IF or ELSE. If it is an simple ELSE clause the statements lying between the ELSE and the END statement are processed. If it is an ELSE IF clause, the specified condition is evaluated and if true the statements lying between that ELSE IF and the next ELSE IF or ELSE or END are processed. This process continues until the IF construct is completed.

The syntactical element <condition> is defined and discussed in Section 6.1 below.

The statements within an IF construct may be any action or control statements, including nested control constructs.

Here is an example of the use of the simple IF construct:

```
IF INERTIAL_POSITION(3) > 0
     <statement>
     <statement>
END IF
```

In this example, certain statements are executed if and only if the spacecraft is currently over the northern hemisphere.

Here is a more elaborate example of an IF construct:

```
IF MAJOR_MODE = 301
     <statement>
ELSE IF MAJOR_MODE = 302
     <statement>
ELSE IF MAJOR_MODE = 303
     <statement>
ELSE
     <statement>
END IF
```

In this example different statements are executed depending on whether mode is 301, 302, or 303, and if mode is none of them a fourth statement is executed.

The modified EVERY construct is useful when a TIMELINER sequence needs to perform certain actions at intervals for some period of time or until some condition occurs. Returning to the analogy of a gate across a circular road, a BEFORE or WITHIN clause establishes a path out of the circle that will be taken upon fulfillment of the BEFORE or WITHIN.

Here is an example of the modified EVERY construct:

```
EVERY 1
   WITHIN 300
      <statement>
      <statement>
END EVERY
EVERY 2
   BEFORE MAJOR_MODE /= 302
      <statement>
      <statement>
END EVERY
```

In this case the sequence processes two statements every one second for 5 minutes. Execution then moves to the second EVERY construct, which executes different statements every 2 seconds until major mode is no longer 302.

The user may find it useful to nest an EVERY construct inside other constructs such as WHEN or WHENEVER. Consider the example:

```
WHENEVER JET_FAIL = ON
      EVERY 0.05
         WITHIN 0.5
            PRINT AVAILABLE_JETS
      END EVERY
END WHENEVER
```

Whenever a jet failure occurs the nested EVERY construct will print the jet availability matrix every 50 ms. for half a second. Eleven printings will occur.

## 4.4 The IF Construct

The IF statement is used to choose among actions -- at a particular point in time. The IF statement initiates an IF construct.

Unlike the WHEN, WHENEVER, and WAIT constructs, an IF construct is not tied in any way to the passage of time. The choice embodied in the IF statement is processed all at once.

An IF construct in its simplest form consists of an IF statement, an END statement, and the statements that lie between. An IF construct may optionally contain multiple ELSE IF clauses and an ELSE clause.

The ELSE statement may alternatively be of the form:

```
ELSE  IF  <condition>
```

where <condition> is as described in Section 6.1 below.

In this case the statements following the ELSE statement are executed if the <condition> specified in the IF statement, and in any preceding ELSE IF statements, evaluate to be false, and if the <condition> in the present ELSE IF statement evaluates to true.

A particular IF construct may contain any number of ELSE IF statements, but at most one plain ELSE statement.

### 4.9 The END Statement

The END statement is used to conclude control constructs, namely the WHEN (4.1), WHENEVER (4.2), EVERY (4.3), and IF (4.4) constructs. Please consult the referenced sections of this User's Guide for an explanation of how END is used.

The form of the END statement is:

```
END  [<construct_type>]
```

where <construct_type> is the single word WHEN, WHENEVER, EVERY, or IF. The <construct_type>, if included, must agree with the type of the construct being concluded.

### 4.10 The WAIT Statement

The WAIT statement is used to introduce a pause into a TIMELINER sequence. Only the particular sequence (or subsequence) containing the WAIT statement pauses.

The WAIT statement is of the form:

```
WAIT  <time_interval>
```

where <time_interval> is expressed as a numeric literal or as a variable name (see Section 6.3 below). If the time interval is a variable, it will be evaluated only once, at the beginning of the wait.

### 4.11 The CALL Statement

The CALL statement is used to invoke a subsequence, as described in Section 2.3 above. A CALL statement may be present in any TIMELINER sequence or subsequence, and may be nested within a control construct.

The CALL statement is of the form:

```
CALL  <name_of_subsequence>
```

### 4.5 The BEFORE Statement

The BEFORE statement is a "modifier" used to create an alternative condition as part of a WHEN (4.1.2), WHENEVER (4.2.2), or EVERY (4.3.2) construct. Please consult the referenced sections of this User's Guide for an explanation of how BEFORE is used.

The form of the BEFORE statement is:

```
      BEFORE  <condition>
```

where <condition> is as described in Section 6.1 below.

### 4.6 The WITHIN Statement

The WITHIN statement is a "modifier" used to create an alternative condition as part of a WHEN (4.1.2), WHENEVER (4.2.2), or EVERY (4.3.2) construct. Please consult the referenced sections of this User's Guide for an explanation of how WITHIN is used.

The form of the WITHIN statement is:

```
      WITHIN  <time_interval>
```

where <time_interval> is as described in Section 6.3 below.

### 4.7 The OTHERWISE Statement

The OTHERWISE statement is used as part of a modified WHEN construct to introduce statements that are executed instead of the normal statements in the event that the BEFORE or WITHIN condition arises before or at the same time as the condition given in the WHEN statement itself, as described in Section 4.1.2 of this User's Guide.

The form of the OTHERWISE statement is:

```
      OTHERWISE
```

The line must contain no additional material except comments.

### 4.8 The ELSE Statement

The ELSE statement is used as part of an IF construct to introduce statements to be executed if the original IF condition, or preceding ELSE IF conditions, fail -- as described in Section 4.4 above.

The ELSE statement may consist of the single word "ELSE", standing alone on a line. In this case the statements following the ELSE statement are executed if the <condition> specified in the IF statement, and in any ELSE IF statements that may form part of the construct, evaluate to be false.

## 5.0 ACTION STATEMENTS

This chapter describes the "action" statements that specify the actions to be taken under conditions specified by TIMELINER control statements.

The action statements implemented in the Ada-language version of TIMELINER at the present time are the LOAD statement, the PRINT statement, and the ABORT statement. The dummy action statement ACTION is also available. Further action statements will be added as the need arises.

## 5.1 The LOAD Statement

The LOAD statement is used to load data into a particular variable known to TIMELINER. Two distinct forms of the LOAD statement are implemented. The simple LOAD statement loads a variable from data provided as literals in the LOAD statement. The LOAD FROM statement loads a variable from another variable. These forms are described separately below.

### 5.1.1 The Simple LOAD Statement

The simple LOAD statement is the type normally used to initialize a variable at the start of a run, or to set a variable at some point during the run under conditions specified by means of TIMELINER control statements.

The simple LOAD statement has the form:

```
LOAD  <name_of_variable>  [ALL]  [<scale_factor>]  <data>
```

where <name_of_variable> names any numeric, boolean, or character string variable known to TIMELINER.

The variable to be loaded may be arrayed in up to three dimensions. Subscripts may form part of the <name_of_variable>, in which case they lie inside parentheses, separated by commas. An asterisk subscript ("*") may be used to indicate that multiple array components are to be loaded.

The optional word ALL may be used to indicate that all components of an arrayed variable are to be loaded to the same value. In this case only one piece of data is required or allowed, regardless of how many components are to be loaded. If ALL is not present, the number of data elements provided must correspond to the arrayness of the variable.

The optional element <scale_factor> may be used to specify a scale factor by which numeric data will be multiplied before insertion into the variable. Scale factors are described in Section 6.4 below. Scale factors possess names of the form M_TO_FT, DEG_TO_RAD, etc. If the application contains variables having the same names as these scale factors, these may not be loaded by a LOAD statement.

When encountered, a CALL statement causes control to be transferred to the SUBSEQUENCE header statement that begins the subsequence that is named. The subsequence then executes. Upon completion of the subsequence control is returned to the sequence or subsequence that contains the CALL statement. Execution resumes at the statement following the CALL statement.

Because a subsequence, like all TIMELINER input, is simply a script to be acted out, and because the TIMELINER internal registers containing the status of a sequence (such as the line pointer) are owned by the calling sequence not the subsequence, all subsequences are "reentrant". That is, a subsequence may be called without worrying about whether the same subsequence may still be in execution due to another call from another sequence.

A called subsequence must lie within the same TIMELINER bundle as the sequence or subsequence that contains the CALL statement.

A control construct such as WHEN, WHENEVER, EVERY or IF must be completed within the same sequence or subsequence where it begins. That is, a control construct may not begin in the calling sequence (or subsequence) and end in the called subsequence, or vice versa.

known to TIMELINER, or an "arithmetic combination" of nouns of numeric type, or a "function" of a noun of numeric type.

Here are some examples of valid LOAD FROM statements:

```
LOAD GUID_ACTIVE FROM SIM_FLAG

LOAD FD2(1,2) FROM FD2(2,1)

LOAD DAP_MODE FROM GUID_MODE

LOAD A FROM A * A

LOAD B FROM B * B

LOAD H FROM A + B

LOAD H FROM SQRT OF H
```

Note that the LOAD-FROM capability can be used to perform algebraic computations in the TIMELINER language. Spare scalars or integers may be used as "accumulators" when the computation is complex. Although this capability is tortuous, it may occasionally prove useful. (The last four examples above compute the hypotenuse of a right triangle.)

## 5.2 The PRINT Statement

The PRINT statement is used to print the contents of any variable known to TIMELINER.

The PRINT statement has the form:

```
PRINT  <name_of_variable>
```

where <name_of_variable> names any numeric, boolean, or character string variable known to TIMELINER.

When it is executed the PRINT statement causes the named variable to be printed. An effort is made to print the variable in an easy-to-read format.

## 5.3 The ABORT Statement

The ABORT statement is used to terminate the simulation. It consists of the single word "ABORT", standing alone on a line. When it is executed the ABORT statement causes measures to be taken to terminate the run as soon as possible.

## 5.4 The ACTION Statement

The ACTION statement is a dummy action statement that was created for purposes of checking out TIMELINER. It has been retained because it may

The element <data> denotes the data to be loaded into the named variable. Such data consists of one or more numeric literals, boolean literals, and string literals. The number of data elements provided must correspond to the number required by the variable to be loaded, as subscripted. Normally only one type of data element is required for a given LOAD. However, an Ada "record" type variable may require data of disparate types.

The <boolean_literal> is described below in Section 6.4.1, the <numeric_literal> in Section 6.4.3, and the <string_literal> in Section 6.4.5. But they are probably just what you think they are.

The simple LOAD statement is the only TIMELINER statement that is allowed to occupy multiple lines in a TIMELINER script. This is required because the number of data elements needed to load a large array variable may not fit on a single line. TIMELINER will continue to read new lines until the required number of data items, of appropriate type, have been ingested. If a new line appears to be a completely new statement, then an error message (Cusses 65-67) will be issued to indicate that insufficient LOAD data has been provided. Error messages (Cusses 62-64) are also issued if too many data items are provided.

Here are some examples of valid LOAD statements:

```
LOAD ABORT_FLAG ON                    -- ABORT_FLAG is boolean

LOAD R +1.743883E+6 .1 .1             -- R is 3-vector

LOAD DAP_MODE "AUTO"                  -- DAP_MODE is char string

LOAD FD3(1,*,*) 1 2 3 4 1 2 3 4
                1 2 3 4 1 2 3 4       -- FD3 is 4x4x4 numeric array

LOAD V(1) M_TO_FT 363.3               -- V is vector scaled in f/s

LOAD T1 23/23:23:23                   -- data converted to seconds

LOAD FD1 ALL RAD_TO_DEG .01744        -- all components loaded to 1
```

### 5.1.2 The LOAD-FROM Statement

The other form of the LOAD statement permits a variable to be loaded from another variable, or from an "arithmetic combination" or "function" of other variables. Thus LOAD-FROM is equivalent to an "assignment" statement.

The LOAD-FROM statement has the form:

```
LOAD  <name_of_variable>  FROM  <noun>
```

where <name_of_variable> names any unarrayed numeric, boolean, or character string variable known to TIMELINER. If the variable is arrayed, its subscripts must reduce it to a particular component.

The element <noun> is fully described below in Section 6.4. In brief, a <noun> is any unarrayed numeric, boolean, or character string variable

## 6.0 SYNTACTICAL ELEMENTS

Throughout this User's Guide reference has been made to certain "syntactical elements" such as <condition>, <time_interval>, <noun>, etc. Where these elements were referenced statement examples were given that illustrated these elements, but no formal description was provided. These syntactical elements are defined and discussed in this chapter.

### 6.1 <condition>

The syntactical element <condition> forms part of the WHEN, WHENEVER, BEFORE, IF, and ELSE IF statements.

<condition> is defined as follows:

```
      <condition>  ::=  [<comparison>  [{AND | OR | XOR}  <comparison>]]
```

where <comparison> is as defined is Section 6.2 below. In this version of TIMELINER, at most two comparisons may be logically linked.

The conjunction "AND" may be written as "&". The conjunction "OR" may be written as "|". There is no alternative form for the exclusive-or conjunction.

When a <condition> is evaluated at execution-time, the component comparisons are first evaluated. Then the indicated logical combination of comparisons is evaluated.

Here are some examples of valid conditions:

```
      TIME > 3600

      X > Y AND Y > Z + 1

      DAP_MODE = "AUTO"

      NAV_ACTIVE = ON OR GUID_ACTIVE = ON

      LOOK_ANGLE <= ARCSIN OF .707
```

Note that it is legal to employ the "null" <condition> that is simply a blank. A null <condition> always passes. This may be useful in cases where the user desires to set up a structure of WHEN's and other constructs without yet knowing the conditions that are relevant.

### 6.2 <comparison>

The syntactical element <comparison> forms part of the element <condition>.

prove useful as a place-holder or an indication of an action that would be performed if the capability existed.

The ACTION statement has the form:

```
      ACTION  <name_of_action>
```

where <name_of_action> is a single string of alphanumeric characters without imbedded blanks -- of maximum length 32.

When executed, the ACTION statement is simply printed.  No other action is performed.

<comparison> is defined as follows:

```
      <comparison>  ::=  <noun>  {= | /= | > | >= | < | <=}  <noun>
```

where <noun> is as defined is Section 6.3 below.

The nouns on the two sides of the comparison must be compatible. That is, both must be numeric, both must be boolean, or both must be of character string type.

All six of the comparators listed above are legal if the nouns are of numeric type. If the nouns are of boolean or character string type, only the equals (=) and not equals (/=) comparators are permitted.

### 6.3 <time_interval>

The syntactical element <time_interval> forms part of the EVERY, BEFORE, and WAIT statements, where it specifies an interval of time, in seconds, that is to be used as appropriate for each type of statement.

<time_interval> is defined as follows:

```
      <time_interval>  ::=  <numeric_noun>
```

where <numeric_noun> is a <noun> of numeric type, either a <numeric_literal> (6.4.3), a <numeric_variable> (6.4.4), or an <arithmetic_combo> (6.4.7). For a more complete description of these noun types see the referenced section.

For use, a <time_interval> is considered to be measured in seconds. If the interval is specified as a <numeric_literal> a compile-time error message (Cuss 72) is issued if the literal is negative. If the interval is specified as a <numeric_variable> or an <arithmetic_combo> a zero time interval will result if the noun evaluates as a negative number at execution-time.

### 6.4 <noun>

The syntactical element <noun> forms part of the element <comparison>. The <noun> also appears as part of a LOAD-FROM statement (Section 5.1.2).

<noun> is defined as follows:

```
      <noun>  ::=  {<boolean>  |  <numeric>  |  <string>}
```

where

```
      <boolean>  ::=  {<boolean_literal>  |  <boolean_variable>}

      <numeric>  ::=  {<numeric_literal>  |  <numeric_variable>  |
                               <arithmetic_combo>}

      <string>   ::=  {<string_literal>  |  <string_variable>}
```

In simple English, a <noun> is an unarrayed literal or variable of numeric, boolean or string type. The <arithmetic_combo> is a little more elaborate. The seven noun types are defined and discussed in the following subsections.

### 6.4.1 <boolean_literal>

The boolean literal is defined as follows:

```
<boolean_literal>  ::=  {ON | OFF | TRUE | FALSE}
```

Naturally ON is equivalent to TRUE and OFF is equivalent to FALSE. The alternative forms have been provided as a service to the user.

### 6.4.2 <boolean_variable>

The <boolean_variable> is defined as any unarrayed variable of boolean type known to TIMELINER. In Ada terms, certain other variables of bi-polar type may be implemented as boolean variables. Such a variable may be "unarrayed" either because it is declared as unarrayed, or because its subscripts reduce it to a single component.

### 6.4.3 <numeric_literal>

The <numeric_literal> is a string of characters that can be converted into a number. As such, it may be a scalar (i.e. it has a decimal point), or an integer (it does not). It may or may not contain an exponent.

In brief, a <numeric_literal> is any string that can be converted to an Ada 8-byte float or 4-byte integer by means of the GET routines provided as part of the package TEXT_IO.

In addition, TIMELINER provides several forms not accepted by TEXT_IO, as follows:

* Numbers that begin with a decimal point, or with a sign followed immediately by a decimal point.

* Numbers that use the familiar DDD/HH:MM:SS.SS format often used to express a time in terms of days, hours, minutes and seconds. Numbers in the formats MM:SS.SS and HH:MM:SS.SS are also accepted. Note that the various fields are not magnitude checked. Thus the forms 96:00:12.1 and 4/00:00:12.1 are both legal, and they are equivalent. Only the "seconds" field may contain a decimal point. Although designed for expressing times, this format might also be used for expressing angles in terms of degrees, minutes, and seconds. TIMELINER does not interpret the slash symbol as a "divide" operator because it does not have spaces on both sides.

Here are some examples of valid numeric literals (in each case the right hand column indicates how the number is internally stored by TIMELINER):

| | |
|---|---|
| 1000 | 1.00000000000000E+03 |
| -.75 | -7.50000000000000E-01 |
| 1.23456789012345E+67 | 1.23456789012345E+67 |
| 11:11 | 6.71000000000000E+02 |
| 364/23:59:59.999 | 3.15359999990000E+07 |
| 1_222_333_444 | 1.22233344400000E+09 |
| 2#11111111# | 2.55000000000000E+02 |
| 16#AA# | 1.70000000000000E+02 |
| -9999999999999995.0 | -1.00000000000000E+16 |

Note that TIMELINER stores all numeric literals in a table of 64-bit floating point numbers. If the user expresses a number using more that 15 significant digits, precision will be lost (as in the final example above).

Note that literals expressed as integers (i.e. with no decimal point) are limited to the range -2147483648 to +2147483647 for reasons related to the implementation of Ada. This can be worked around by adding ".0" to convert the number to floating point.

### 6.4.4 <numeric_variable>

The <numeric_variable> is defined as any unarrayed variable of numeric type known to TIMELINER. Such a variable may be "unarrayed" either because it is declared as unarrayed, or because its subscripts reduce it to a single component.

### 6.4.5 <string_literal>

The string literal is defined as any character string, with or without blanks, that is included between double or single quotation marks. When a <string_literal> is used in a <comparison> any leading and trailing blanks are first removed.

### 6.4.6 <string_variable>

The <string_variable> is defined as any unarrayed variable of string type known to TIMELINER. Such a variable may be "unarrayed" either because it is declared as unarrayed, or because its subscripts reduce it to a single component. When a <string_variable> is used in a <comparison> any leading and trailing blanks are first removed.

### 6.4.7 <arithmetic_combo>

The <arithmetic_combo> is a noun of numeric type that is formed as an arithmetic combination of two numeric literals or variables, or as a recognized function of a numeric literal or variable.

That is, the allowable forms of an <arithmetic_combo> are

```
<noun>  <arithmetic_operator>  <noun>
```

and

```
<function>  OF  <noun>
```

An <arithmetic_operator> may be one of the following:

| | |
|---|---|
| + | addition |
| - | subtraction |
| * | multiplication |
| / | division |
| ** | exponentiation |
| MOD | the modulo operation |

A <function> may be one of the following:

| | |
|---|---|
| ABS | absolute value |
| SQRT | square root |
| SIN \| SINE | trigonometric sine |
| COS \| COSINE | trigonometric cosine |
| TAN \| TANGENT | trigonometric tangent |
| ARCSIN \| ARCSINE | trigonometric arcsine |
| ARCCOS \| ARCCOSINE | trigonometric arccossine |
| ARCTAN \| ARCTANGENT | trigonometric arctangent |

The following are examples of valid arithmetic combinations:

```
X + Y
FD3(1,1,1) + 1.23456789012345E+67
SQRT OF 2
ARCTAN OF RESOLVER(3)
```

The <arithmetic_combo> can be used any place that a <numeric_literal> or a <numeric_variable> may be used, except that it may not be used to form another <arithmetic_combo>.

## 6.5 <scale_factor>

TIMELINER contains the capability of specifying a scale factor as part of a simple LOAD statement (see Section 5.1.1). Such scale factors are of the form "xx_TO_xx". In the present Ada-language implementation of TIMELINER the following scale factors are allowed:

| | |
|---|---|
| M_TO_FT | meters to feet |
| FT_TO_M | feet to meters |
| DEG_TO_RAD | degrees to radians |
| RAD_TO_DEG | radians to degrees |

Please let the authors know of any other scale factors that would be convenient.

## 7.0 TIMELINER ERROR MESSAGES

At compile-time TIMELINER issues error messages designed to facilitate the user's effort to create an executable TIMELINER script. TIMELINER error messages are called "cusses". This chapter describes the various cusses that may be issued.

TIMELINER makes an effort to keep going, even if there are many errors, in the hope of giving the user enough information to eliminate all errors in a single pass. There is no maximum number of errors that will cause TIMELINER to quit parsing the input stream.

The authors of TIMELINER would like to be told if you encounter cases where the error messages that are issued seem misleading, or where the existence of one error avoidably masks the existence of another.

The following error messages are issued by TIMELINER:

### Cuss 1 -- unrecognized_directive

The compile-time error message

```
UNRECOGNIZED TIMELINER DIRECTIVE STATEMENT:
```

is issued if a "directive" statement is faulty. The word "directive" refers to a capability installed for checkout purposes that is of no interest to users.

### Cuss 2 -- print_level_not_numeric_lit

The compile-time error message

```
PRINT LEVEL CAN ONLY BE SET TO A NUMERIC LITERAL:
```

is issued if a "directive" statement intended to change TIMELINER's compile-time "print level" is defective. The word "directive" refers to a capability installed for checkout purposes that is of no interest to users.

### Cuss 3 -- too_many_lines

The compile-time error message

```
TOO MANY LINES IN SCRIPT -- MAXIMUM IS nst
```

is issued if the number of lines (i.e. statements) in the script exceeds the capacity of the TIMELINER tables. The number given in the cuss is the maximum number of statements that can currently be digested. This number can be increased by changing the parameter "nst" in the module TL_DATA_COM and recompiling all TIMELINER modules. For a given TIMELINER script, the usage of the table space available for statements is shown as part of the "file usage summary" printed at the end of the compile-time listing.

### Cuss 4 -- too_many_words

The compile-time error message

```
TOO MANY WORDS ON LINE -- MAXIMUM IS nlw
```

is issued if the number of words on a line of script exceeds the maximum that is permitted. This number is set by parameter "nlw" in the module TL_INIT_COM. Please do not change this parameter without first checking with the author. The only way this cuss can occur within proper syntax is if a LOAD statement contains too many items of data on a single line. In this case the limitation can be worked around by allowing the LOAD data to spill onto additional lines.

### Cuss 5 -- too_many_chars

The compile-time error message

```
TOO MANY CHARACTERS IN WORD -- MAX IS nwl -- WORD TRUNCATED:
```

is issued if the number of characters in a word exceeds the maximum that is permitted. This number is set by parameter "nwl" in the module TL_DATA_COM. Please do not change this parameter without first checking with the author.

### Cuss 9 -- too_many_slits

The compile-time error message

TOO MANY STRING LITERALS IN SCRIPT -- MAXIMUM IS nsl

is issued if the number of string literals (defined in Section 6.4.5) exceeds the capacity of the TIMELINER tables. The number given in the cuss is the maximum number of string literals that can currently be digested. This number can be increased by changing the parameter "nsl" in the module TL_DATA_COM and recompiling all TIMELINER modules. For a given TIMELINER script, the usage of the table space available for string literals is shown as part of the "file usage summary" printed at the end of the compile-time listing.

### Cuss 10 -- too_many_nouns

The compile-time error message

TOO MANY NOUNS IN SCRIPT -- MAXIMUM IS nnn

is issued if the number of nouns (see Section 6.3) exceeds the capacity of the TIMELINER tables. The number given in the cuss is the maximum number of nouns that can currently be digested. This number can be increased by changing the parameter "nnn" in the module TL_DATA_COM and recompiling all TIMELINER modules. For a given TIMELINER script, the usage of the table space available for nouns is shown as part of the "file usage summary" printed at the end of the compile-time listing.

### Cuss 11 -- too_many_words_for_type

The compile-time error message

TOO MANY WORDS FOR STATEMENT TYPE -- MAXIMUM IS n

is issued if the number of words on a line is greater than the maximum number of words that may be required for a statement of a given type. The number "n" is that maximum. This cuss therefore indicates that the statement contains a syntactical error. Additional cusses may further illuminate the mistake.

### Cuss 12 -- too_few_words_for_type

The compile-time error message

TOO FEW WORDS FOR STATEMENT TYPE -- MINIMUM IS n

is issued if the number of words on a line is fewer than the minimum number of words that may be required for a statement of a given type. The number "n" is that minimum. This cuss therefore indicates that the statement contains a syntactical error. Additional cusses may further illuminate the mistake.

**Cuss 6 -- too_many_blocks**

The compile-time error message

TOO MANY SEQUENCEs AND SUBSEQUENCEs IN SCRIPT -- MAXIMUM IS nsq

is issued if the number of sequences and subsequences in the script exceeds the capacity of the TIMELINER tables. The number given in the cuss is the maximum number of sequences and subsequences that can currently be digested. This number can be increased by changing the parameter "nsq" in the module TL_DATA_COM and recompiling all TIMELINER modules. For a given TIMELINER script, the usage of the table space available for sequences and subsequences is shown as part of the "file usage summary" printed at the end of the compile-time listing.

**Cuss 7 -- too_many_blits**

The compile-time error message

TOO MANY BOOLEAN LITERALS IN SCRIPT -- MAXIMUM IS nbl

is issued if the number of boolean literals (defined in Section 6.4.1) exceeds the capacity of the TIMELINER tables. The number given in the cuss is the maximum number of boolean literals that can currently be digested. This number can be increased by changing the parameter "nbl" in the module TL_DATA_COM and recompiling all TIMELINER modules. For a given TIMELINER script, the usage of the table space available for boolean literals is shown as part of the "file usage summary" printed at the end of the compile-time listing.

**Cuss 8 -- too_many_nlits**

The compile-time error message

TOO MANY NUMERIC LITERALS IN SCRIPT -- MAXIMUM IS nnl

is issued if the number of numeric literals (defined in Section 6.4.3) exceeds the capacity of the TIMELINER tables. The number given in the cuss is the maximum number of numeric literals that can currently be digested. This number can be increased by changing the parameter "nnl" in the module TL_DATA_COM and recompiling all TIMELINER modules. For a given TIMELINER script, the usage of the table space available for numeric literals is shown as part of the "file usage summary" printed at the end of the compile-time listing.

### Cuss 17 -- construct_open

The compile-time error message

> CONSTRUCT OPEN AT CLOSE OF SEQUENCE OR SUBSEQUENCE:

is issued if a control construct such as WHEN, WHENEVER, EVERY or IF remains open when a CLOSE SEQUENCE or CLOSE SUBSEQUENCE statement is encountered. All constructs must be closed (using END) before the sequence or subsequence that contains them can be closed. The material printed after the colon indicates the type of construct that is open. The cuss probably means that the END statement of the construct indicated, or of some other construct nested inside the indicated construct, has been omitted.

### Cuss 18 -- nesting_too_deep

The compile-time error message

> CONSTRUCTS NESTED TOO DEEPLY -- MAXIMUM DEPTH IS nll

is issued if the depth of construct nesting exceeds the capacity of the TIMELINER tables. The number given in the cuss is the maximum number of levels of nesting that can currently be handled. This number can be increased by changing the parameter "nll" in the module TL_DATA_COM and recompiling all TIMELINER modules. For a given TIMELINER script, the usage of the depth available for construct nesting is shown as part of the "file usage summary" printed at the end of the compile-time listing.

### Cuss 19 -- start_with_bundle

The compile-time error message

> INPUT STREAM MUST BEGIN WITH 'BUNDLE' STATEMENT

is issued if the first statement in the input stream is not a BUNDLE header statement. Since all TIMELINER input must belong to some "bundle" the first statement must always be a BUNDLE header. The correct organization of a TIMELINER script into hierarchical blocks is explained in Chapter 2 of this User's Guide.

### Cuss 20 -- end_with_close_bundle

The compile-time error message

> INPUT STREAM MUST END WITH 'CLOSE BUNDLE' STATEMENT

is issued if the last statement in the input stream is not a CLOSE BUNDLE statement. Since all TIMELINER input must belong to some "bundle" the last statement must always be a CLOSE BUNDLE statement. The correct organization of a TIMELINER script into hierarchical blocks is explained in Chapter 2 of this User's Guide.

**Cuss 13 -- wrong_number_of_words**

The compile-time error message

WRONG NUMBER OF WORDS FOR STATEMENT TYPE

is issued if the number of words on a line is a number than should never be the case for a statement of a given type. This cuss therefore indicates that the statement contains a syntactical error. Additional cusses may further illuminate the mistake.

**Cuss 14 -- paren_open**

The compile-time error message

LINE ENDS WITH PARENTHESES OPEN

is issued if a line of script contains an unequal number of open-parentheses and close-parentheses. Since the only function of parenthese in the TIMELINER language is to hold variable subscripts, this cuss indicates that a subscript field is faulty.

**Cuss 15 -- quote_open**

The compile-time error message

LINE ENDS WITH QUOTATION MARKS OPEN

is issued if a line of script contains an unequal number of (double or single) quotation marks. Since the only function of quotation marks in the TIMELINER language is to denote a string literal, this cuss probably means that a string literal is faulty.

**Cuss 16 -- block_open**

The compile-time error message

SEQUENCE/SUBSEQUENCE OPEN AT CLOSE OF BUNDLE

is issued if a sequence or subsequence remains unclosed when a CLOSE BUNDLE statement is encountered. All sequences and subsequences must be closed (using CLOSE) before the bundle that contains them can be closed. The correct organization of a TIMELINER script into hierarchical blocks is explained in Chapter 2 of this User's Guide.

### Cuss 25 -- close_mismatched

The compile-time error message

> 'CLOSE' STATEMENT DOES NOT CORRESPOND TO TYPE OF OPEN BLOCK:

is issued if the type indicated by the second word in a CLOSE statement does not match the type of the innermost block (bundle, sequence, or subsequence) that is open. The material after the colon is the erroneous word.

### Cuss 26 -- close_name_mismatch

The compile-time error message

> NAME IN 'CLOSE' STATEMENT DOES NOT MATCH NAME OF OPEN BLOCK:

is issued if the name that may be included as the third word in a CLOSE statement does not match the name of the innermost block (bundle, sequence, or subsequence) that is open. The material after the colon is the erroneous word.

### Cuss 27 -- close_unknown

The compile-time error message

> 'CLOSE' TYPE UNKNOWN -- MUST BE BUNDLE, SEQUENCE, OR SUBSEQUENCE:

is issued if the type indicated by the second word in a CLOSE statement is not recognized. The most likely cause of this cuss is that the user mistyped the word BUNDLE, SEQ[UENCE], or SUB[SEQ[UENCE]]. The material after the colon is the erroneous word.

### Cuss 28 -- line_type_unknown

The compile-time error message

> STATEMENT TYPE NOT RECOGNIZED:

is issued if the type indicated by the first word on a line of script does not correspond to any legal TIMELINER statement type.

**Cuss 21 -- stat_outside_bundle**

The compile-time error message

STATEMENT INAPPROPRIATE BECAUSE NO BUNDLE IS OPEN:

is issued if any statement (other than a BUNDLE header or CLOSE) is encountered that does not lie inside a "bundle". The correct organization of a TIMELINER script into hierarchical blocks is explained in Chapter 2 of this User's Guide.

**Cuss 22 -- stat_outside_sequence**

The compile-time error message

STATEMENT INAPPROPRIATE BECAUSE NO SEQ OR SUBSEQ IS OPEN:

is issued if any control or action statement is found outside the confines of a sequence or subsequence. The correct organization of a TIMELINER script into hierarchical blocks is explained in Chapter 2 of this User's Guide.

**Cuss 23 -- bundles_nested**

The compile-time error message

BUNDLES MAY NOT BE NESTED

is issued if a BUNDLE header statement is encountered when an existing bundle has not been closed. Only one bundle may be open at a time. The correct organization of a TIMELINER script into hierarchical blocks is explained in Chapter 2 of this User's Guide.

**Cuss 24 -- sequences_nested**

The compile-time error message

SEQUENCES/SUBSEQUENCES MAY NOT BE NESTED

is issued if a SEQUENCE or SUBSEQUENCE header statement is encountered when an existing sequence or subsequence has not been closed. Only one sequence/subsequence may be open at a time. The correct organization of a TIMELINER script into hierarchical blocks is explained in Chapter 2 of this User's Guide.

**Cuss 33 -- otherwise_outside**

The compile-time error message

'OTHERWISE' VALID ONLY INSIDE A 'WHEN' CONSTRUCT

is issued if an OTHERWISE statement is encountered outside of a WHEN construct. For an explanation of the correct use of the OTHERWISE statement within a WHEN construct see Section 4.1.2 of this User's Guide.

**Cuss 34 -- otherwise_already**

The compile-time error message

'WHEN' CONSTRUCT ALREADY HAS AN 'OTHERWISE'

is issued if a second OTHERWISE statement is encountered inside a WHEN construct. For an explanation of the correct use of the OTHERWISE statement within a WHEN construct see Section 4.1.2 of this User's Guide.

**Cuss 35 -- otherwise_meaningless**

The compile-time error message

'OTHERWISE' MEANINGLESS BECAUSE THERE IS NO 'BEFORE' OR 'WITHIN'

is issued if an OTHERWISE statement in encountered inside a WHEN construct that does not contain a BEFORE or WITHIN statement. For an explanation of the correct use of the OTHERWISE statement within a WHEN construct see Section 4.1.2 of this User's Guide.

**Cuss 36 -- else_outside**

The compile-time error message

'ELSE' STATEMENT MUST BE INSIDE AN 'IF' CONSTRUCT

is issued if an ELSE or ELSE IF statement is encountered outside of an IF construct. For an explanation of the correct use of the ELSE and ELSE IF statements within an IF construct see Section 4.4.

### Cuss 29 -- keystroke_unknown

The compile-time error message

KEYSTROKE NOT RECOGNIZED:

is issued if a keystroke forming part of a KEY statement is not a keystroke that is legal within the application to which the version of TIMELINER is attached. (Note that the KEY statement is not currently implemented in the Ada-language version of TIMELINER, so this cuss is impossible.)

### Cuss 30 -- modifier_outside

The compile-time error message

'BEFORE/WITHIN' MUST BE INSIDE A 'WHEN/WHENEVER/EVERY' CONSTRUCT

is issued if a BEFORE or WITHIN statement is encountered that lies outside of any WHEN (4.1.2), WHENEVER (4.2.2), or EVERY (4.3.2) construct. For an explanation of the correct use of the BEFORE and WITHIN statements within such a construct see the indicated section of this User's Guide.

### Cuss 31 -- modifier_already

The compile-time error message

'WHEN/WHENEVER/EVERY' CONSTRUCT ALREADY HAS A 'BEFORE/WITHIN'

is issued if a second BEFORE or WITHIN statement is encountered within a WHEN (4.1.2), WHENEVER (4.2.2), or EVERY (4.3.2) construct. For an explanation of the correct use of the BEFORE and WITHIN statements within such a construct see the indicated section of this User's Guide.

### Cuss 32 -- modifier_misplaced

The compile-time error message

'BEFORE/WITHIN' MUST IMMEDIATELY FOLLOW 'WHEN/WHENEVER/EVERY'

is issued if a BEFORE or WITHIN statement is encountered that is incorrectly placed within a WHEN (4.1.2), WHENEVER (4.2.2), or EVERY (4.3.2) construct. A BEFORE or WITHIN statement must immediately follow the statement that opens the construct. For an explanation of the correct use of the BEFORE and WITHIN statements within a construct see the indicated section of this User's Guide.

**Cuss 41 -- bad_comp_pattern**

The compile-time error message

PATTERN OF COMPARISON EXPRESSION NOT RECOGNIZED:

is issued if the pattern of a <comparison> making up part of a <condition> in a WHEN, WHENEVER, BEFORE, IF, or ELSE IF statement cannot be parsed because of a syntax error. See Section 6.2 for a description and explanation of the syntactical element <comparison>.

**Cuss 42 -- bad_function**

The compile-time error message

FUNCTION NOT RECOGNIZED:

is issued if TIMELINER does not recognize the type of a function. A function type is indicated by the word preceding the "OF" in an <arithmetic_combo>, as described in Section (6.4.7).

**Cuss 43 -- bad_noun**

The compile-time error message

NOUN NOT RECOGNIZED:

is issued if a <noun> forming part of a <comparison> or a <time_interval> or an <arithmetic_combo> or appearing in a LOAD FROM statement, cannot be recognized. Nouns are described and explained in Section 6.4.

**Cuss 44 -- noun_unsuitable**

The compile-time error message

NOUN UNSUITABLE FOR MAGNITUDE COMPARISON:

is issued if a <noun> of boolean or string type is used in conjunction with a magnitude comparison (i.e. <, >, <=, or >=) in a <comparison>, such that the <comparison> cannot be evaluated.

**Cuss 45 -- nouns_incompatible**

The compile-time error message

NOUNS IN COMPARISON ARE INCOMPATIBLE:

is issued if the nouns on each side of a comparator (i.e. =, /=, <, >, <=, or >=) in a <comparison> are incompatible with each other, such that the <comparison> cannot be evaluated.

**Cuss 37 -- end_outside**

The compile-time error message

```
'END' INAPPROPRIATE BECAUSE NO 'WHEN/WHENEVER/EVERY/IF' IS OPEN
```

is issued if an END statement is encountered for which there is no corresponding WHEN (4.1), WHENEVER (4.2), EVERY (4.3), or IF (4.4) construct. For an explanation of the correct use of the END statement to close a construct see the indicated section of this User's Guide.

**Cuss 38 -- end_mismatched**

The compile-time error message

```
'END' STATEMENT DOES NOT CORRESPOND TO TYPE OF OPEN CONSTRUCT
```

is issued if the type given by the (optional) second word of an END statement does not correspond to the type of the currently open WHEN (4.1), WHENEVER (4.2), EVERY (4.3), or IF (4.4) construct. For an explanation of the correct use of the END statement to close a construct see the indicated section of this User's Guide.

**Cuss 39 -- should_use_close**

The compile-time error message

```
USE 'CLOSE' TO TERMINATE A BUNDLE/SEQUENCE/SUBSEQUENCE
```

is issued if the user attempts to use an END statement to close a bundle, sequence, or subsequence block. END ends constructs only. CLOSE closes blocks.

**Cuss 40 -- should_use_end**

The compile-time error message

```
USE 'END' TO TERMINATE A WHEN/WHENEVER/EVERY/IF
```

is issued if the user attempts to use a CLOSE statement to close a WHEN, WHENEVER, EVERY, or IF construct. CLOSE closes blocks only. END ends constructs.

### Cuss 50 -- noun_not_boolean

The compile-time error message

NOUN MUST BE A BOOLEAN VARIABLE OR LITERAL:

is issued if a <noun> given as part of a LOAD-FROM statement, for a boolean LOAD variable, is not of boolean type. After the colon is printed the improper <noun>.

### Cuss 51 -- noun_not_numeric

The compile-time error message

NOUN MUST BE A NUMERIC OR AN ARITHMETIC COMBO:

is issued if a noun used to specify the <time_interval> in an EVERY, WITHIN, or WAIT statement is a <noun> of boolean or string (i.e. non-numeric) type. This cuss is also issued if a <noun> given as part of a LOAD-FROM statement, for a numeric LOAD variable, is not of numeric type. After the colon is printed the improper <noun>.

### Cuss 52 -- noun_not_string

The compile-time error message

NOUN MUST BE A STRING VARIABLE OR LITERAL:

is issued if a <noun> given as part of a LOAD-FROM statement, for a string LOAD variable, is not of string type. After the colon is printed the improper <noun>.

### Cuss 53 -- too_few_subs

The compile-time error message

TOO FEW SUBSCRIPTS FOR THIS VARIABLE:

is issued if an array variable used in a LOAD or PRINT statement, or as a <noun> in any statement, has been entered with too few subscripts. If a variable is subscripted at all, the number of subscripts given must correspond to the number of dimensions in the array. The material following the colon is the faulty variable name.

**Cuss 46 -- noun_combo_mismatch**

The compile-time error message

NOUN INCOMPATIBLE WITH ARITHMETIC COMBO:

is similar to the preceding error message. It is issued if a <noun> of boolean or string type is compared to a noun of <arithmetic_combo> type as part of a <comparison>. An <arithmetic_combo> (see Section 6.4.7) is a numeric noun and may only be compared to other numeric nouns.

**Cuss 47 -- combo_noun_nonnumeric**

The compile-time error message

NOUN IN ARITHMETIC COMBO MUST BE NUMERIC:

is issued if a <noun> of boolean or string type is used inside of an <arithmetic_combo>. An <arithmetic_combo> (see Section 6.4.7) may only involve <nouns> of numeric type.

**Cuss 48 -- noun_arrayed**

The compile-time error message

NOUN IS ARRAYED -- MUST BE SINGLE:

is issued if a <noun> forming part of a <comparison> or a <time_interval> or a LOAD FROM statement is arrayed. Such a noun must be inherently unarrayed, or must be subscripted to indicate only a single component.

**Cuss 49 -- bad_modifier**

The compile-time error message

NOUN HAS MODIFIER INAPPROPRIATE FOR TYPE:

is issued if a boolean or string noun is preceded by a minus sign, or if a numeric or string noun is preceded by the "not" sign "~".

**Cuss 58 -- subseq_not_found**

The compile-time error message

SUBSEQUENCE CALLED ABOVE IS NOT PRESENT IN BUNDLE:

is issued if, at the time a given BUNDLE is closed, one or more of the subsequences called within that BUNDLE are not present in the BUNDLE. The material after the colon is the name of the missing subsequence.

**Cuss 59 -- duplicate_block_name**

The compile-time error message

A SEQUENCE OR SUBSEQUENCE OF THIS NAME ALREADY EXISTS:

is issued if a name given to a sequence or subsequence in a SEQUENCE or SUBSEQUENCE header statement has already been used by another sequence or subsequence within the same BUNDLE. The material after the colon is the sequence or subsequence name that is already taken.

**Cuss 60 -- bad_variable**

The compile-time error message

LOAD/PRINT VARIABLE NOT RECOGNIZED:

is issued if a word that TIMELINER believes is intended to name a variable does not correspond to the name of any variable known to TIMELINER. Since TIMELINER determines that a word is meant to denote a variable by a process of elimination, this cuss may be issued if a word not intended to denote a variable is misspelled, for example if boolean literal TRUE were misspelled TROU. The material following the colon is the faulty word.

**Cuss 61 -- no_load_data**

The compile-time error message

STATEMENT CONTAINS NO LOAD DATA

is issued if a LOAD statement is encountered that contains no data to be loaded into the variable.

**Cuss 54 -- too_many_subs**

The compile-time error message

TOO MANY SUBSCRIPTS FOR THIS VARIABLE:

is issued if a variable used in a LOAD or PRINT statement, or as a <noun> in any statement, has been entered with too many subscripts. If a variable is subscripted at all, the number of subscripts given must correspond to the number of dimensions in the array. This cuss is issued if an unarrayed variable is subscripted at all. The material following the colon is the faulty variable name.

**Cuss 55 -- subs_out_of_range**

The compile-time error message

ONE OR MORE SUBSCRIPTS OUT-OF-RANGE FOR THIS VARIABLE:

is issued if a subscript attached to an array variable lies outside of the permitted range of subscripts for the given dimension of the variable. Note that at present the array elements for a TIMELINER variable must begin with "1". The material following the colon is the faulty variable name.

**Cuss 56 -- sub_field_error**

The compile-time error message

SUBSCRIPT FIELD BADLY FORMATTED:

is issued if an error is detected in the subscript field of a variable used in a LOAD or PRINT statement, or used as a <noun>. The material following the colon is the faulty variable name.

**Cuss 57 -- too_many_calls**

The compile-time error message

TOO MANY SUBSEQUENCE CALLS -- MAXIMUM IS ncl

is issued if the number of subsequence calls in a given BUNDLE exceeds the maximum permitted number. The allowed number of calls is determined by parameter "ncl" in the module TL_PARSER_BODY. Please contact the authors if you need this number to be increased.

**Cuss 66 -- too_little_num_data**

The compile-time error message

TOO FEW NUMERIC LITERALS PROVIDED -- NEED n

is issued if a LOAD statement is encountered that contains fewer numeric literals than are required to load the variable in question, as subscripted. The number given is the number of numeric literals that are required.

**Cuss 67 -- too_little_str_data**

The compile-time error message

TOO FEW STRING LITERALS PROVIDED -- NEED n

is issued if a LOAD statement is encountered that contains fewer string literals than are required to load the variable in question, as subscripted. The number given is the number of string literals that are required.

**Cuss 68 -- loadfrom_arrayed**

The compile-time error message

'LOAD-FROM' VARIABLE IS ARRAYED -- MUST BE SINGLE:

is issued if a variable to be loaded by a LOAD-FROM statement (see Section 5.1.2) is arrayed. Such a variable must be unarrayed or subscripted such as to name a particular element. The arrayed variable is printed after the colon.

**Cuss 69 -- loadfrom_mismatch**

The compile-time error message

'LOAD-FROM' VARIABLE AND NOUN HAVE DIFFERENT TYPES

is issued if a variable to be loaded by a LOAD-FROM statement (see Section 5.1.2) and the <noun> that names the variable whose contents are to be used have different types. This cuss relates to the TIMELINER types (boolean, numeric, and string) and not to Ada-language types.

**Cuss 62 -- too_much_boo_data**

The compile-time error message

TOO MANY BOOLEAN LITERALS PROVIDED -- NEED n

is issued if a LOAD statement is encountered that contains more boolean literals than are required to load the variable in question, as subscripted. The number given is the number of boolean literals that are required.

**Cuss 63 -- too_much_num_data**

The compile-time error message

TOO MANY NUMERIC LITERALS PROVIDED -- NEED n

is issued if a LOAD statement is encountered that contains more numeric literals than are required to load the variable in question, as subscripted. The number given is the number of numeric literals that are required.

**Cuss 64 -- too_much_str_data**

The compile-time error message

TOO MANY STRING LITERALS PROVIDED -- NEED n

is issued if a LOAD statement is encountered that contains more string literals than are required to load the variable in question, as subscripted. The number given is the number of string literals that are required.

**Cuss 65 -- too_little_boo_data**

The compile-time error message

TOO FEW BOOLEAN LITERALS PROVIDED -- NEED n

is issued if a LOAD statement is encountered that contains fewer boolean literals than are required to load the variable in question, as subscripted. The number given is the number of boolean literals that are required.

## 8.0 TIMELINER MAINTENANCE

Under the heading "TIMELINER maintenance" are included those functions that may occasionally be necessary to adapt the version of TIMELINER dedicated to a particular application to the needs of that application.

This chapter has two parts. The first discusses how to change the sizes of the tables used by TIMELINER to store executable code. The second discusses how to make TIMELINER aware of the variables that form part of the application.

## 8.1 HOW TO CHANGE THE SIZE OF THE TIMELINER TABLES

TIMELINER works by analyzing the raw script input by the user and placing the information contained in the script in a set of tables. The tabulated script information for each bundle (Section 2.1) is then written into a file. At execution-time, the information is read from the file into an identical set of tables, where it is executed.

The size of the TIMELINER tables is determined by a series of parameters. The following table states the name of the parameter, the Ada module where the parameter is defined, and the function of each parameter:

| | | |
|---|---|---|
| nsq | TL_DATA_COM | number of sequences and subsequences |
| nst | TL_DATA_COM | number of statements |
| nsd | TL_DATA_COM | number of pieces of statement data |
| nlv | TL_DATA_COM | allowed levels of construct nesting |
| nbl | TL_DATA_COM | number of boolean literals |
| nnl | TL_DATA_COM | number of numeric literals |
| nsl | TL_DATA_COM | number of string literals |
| nnn | TL_DATA_COM | number of nouns |
| ncl | TL_INIT_COM | number of subseq calls in a bundle |

Each BUNDLE's usage of the tables sized by these parameters is printed as part of the "file usage summary" that forms part of the compile-time listing of a TIMELINER script.

A particular embodiment of the Ada-language version of TIMELINER may require varying amounts of storage, depending upon the length and level of complexity of the scripts that are required. Therefore it is anticipated that a particular TIMELINER application will need to tailor the table sizes controlled by the parameters listed above to meet its particular needs.

**Cuss 70 -- bad_load_input**

The compile-time error message

    'LOAD' INPUT MUST BE BOOLEAN, NUMERIC OR STRING LITERAL:

is issued if an item of data in a simple LOAD statement is not either a <boolean_literal>, a <numeric_literal>, or a <string_literal>. The offending item is given after the colon.

**Cuss 71 -- scale_factor_improper**

The compile-time error message

    SCALE FACTOR IMPROPER BECAUSE VARIABLE IS NOT NUMERIC:

is issued if a scale factor is included as part of a LOAD statement in a case where the variable to be loaded is not of numeric type. The material after the colon is the name of the non-numeric variable.

**Cuss 72 -- negative_time_interval**

The compile-time error message

    'EVERY/WITHIN/WAIT' TIME INTERVAL IS NEGATIVE:");

is issued if the <time_interval> specified in an EVERY, WITHIN, or WAIT statement is a negative numeric literal. If the <time interval> is given in the form of a variable no cuss can be issued, of course, because the value of the variable is unknown at compile time. A negative time interval discovered at run-time will result in a zero interval.

**VAR_VAL_TYPE**

The variable type. The types that the present version of TIMELINER is equipped to handle are specified by the following names:

| | |
|---|---|
| OF_SCALAR_SINGLE | 32-bit float, 6 digit precision |
| OF_SCALAR_DOUBLE | 64-bit float, 12 digit precision |
| OF_INT8 | 8-bit integer |
| OF_INT16 | 16-bit integer |
| OF_INT32 | 32-bit integer |
| OF_BOOLEAN | 1-bit boolean |
| OF_BOOL32 | 32-bit boolean |
| OF_STRING | 40-character string, 320-bits |

New types may be defined by the user in package TL_VAR_TYPES as explained below.

**TLINER_VAL_TYPE**

The TIMELINER type that is most appropriate for the variable in question. The choices are NUMBER_TYPE, LOGICAL_TYPE, and STRING_TYPE, corresponding to the <noun> types <numeric>, <boolean>, and <string> that are described in Section 6.4 of this document.

**NUMB_DIMENSIONS**

The number of dimensions associated with a variable. This number must lie between 0 (un-arrayed variable), and 3 (three-dimensional array).

**DIM1**

The size of the first dimension of a variable arrayed in one, two, or three dimensions. The variable's first subscript must fall into the range between 1 and this number.

**DIM2**

The size of the second dimension of a variable arrayed in two, or three dimensions. The variable's second subscript must fall into the range between 1 and this number.

**DIM3**

The size of the third dimension of a variable arrayed in three dimensions. The variable's third subscript must fall into the range between 1 and this number.

**VAR_SYS_ADDR**

The address, in a particular processor, at which the variable resides. If the variable appears in more than one package specification, this address entry must be prefaced by the package name.

*In the single-processor (functional) version associated of the CSDL real-time SSF DMS testbed, the variable list is located in the package TL_VAR_TYPES found in the file TL_VAR_TYPES_SP_S.ADA.*

## 8.2 HOW TO MAKE TIMELINER AWARE OF APPLICATION VARIABLES

At the core of TIMELINER lies a complex set of logic that makes TIMELINER aware of the variables that form part of the application to which a version of TIMELINER is attached.

The TIMELINER logic that provides access to variables is referred to as the "low-level" TIMELINER logic. The changes required to adapt TIMELINER to a particular application are confined to this low-level logic. TIMELINER's "uppper-level" logic, which implements the language features, is the same for all applications.

The TIMELINER low-level logic is accessed by means of three separate functions:

| | |
|---|---|
| **GRAB_VAR** | Obtains the value of a (un-arrayed) variable for use by a control statement in making a decision. |
| **LOAD_VAR** | Loads the variable (which may be arrayed) using data supplied by the user. |
| **DISPLAY_VAR** | Prints (or otherwise displays) the value of a variable (which may be arrayed). |

In order to perform these three functions with respect to a particular variable, TIMELINER requires that the variable be entered into the TIMELINER "variable list". This process consists of three steps:

* The package specification in which the variable resides must be known to the TIMELINER compilation unit. This is accomplished by 'with'ing the package specification at the top of the compilation unit that contains the variable list.

* The variable must be entered alphabetically into the variable list, including the attributes described below.

* The variable LIST_SIZE must be adjusted to reflect the actual number of variables in the variable list.

The variable attributes for each variable contained in the variable list are:

| | |
|---|---|
| **VAR_NAME** | A character string representing the name of the variable in question. The variable list must be structured in alphabetical order, according to VAR_NAME. In the present version of TIMELINER, VAR_NAME is a string of 40 characters. If the variable appears in more than one package specification, the variable name should be prefaced by the package name. |

To incorporate the new type into TIMELINER, the enumeration type VARIABLE_VALUE_TYPE must be modified to include the new type, for example:

```
type VARIABLE_VALUE_TYPE is  (OF_SCALAR_SINGLE,
                              OF_SCALAR_DOUBLE,
                              OF_INT8,
                              OF_INT16,
                              OF_INT32,
                              OF_BOOLEAN
                              OF_BOOL8,
                              OF_BOOL32,
                              OF_STRING);
```

The number of 8-bit bytes allocated for storage of the newly defined variable type is then represented in NUMB_STORAGE_BYTES, as follows:

```
NUMB_STORAGE_BYTES : VAR_TYPE_SIZE := (OF_SCALAR_SINGLE => 4,
                                       OF_SCALAR_DOUBLE => 8,
                                       OF_INT8          => 1,
                                       OF_INT16         => 2,
                                       OF_INT32         => 4,
                                       OF_BOOL8         => 1,
                                       OF_BOOL32        => 4,
                                       OF_STRING        => 40)
```

**WARNING:** TIMELINER is very dependent on the specific types defined by the enumeration type VARIABLE_VALUE_TYPE and further described in NUMB_STORAGE_BYTES. If new enumeration literals are added to this list or if the names of the enumeration literals are changed, the code in package TL_OBJ_OPERATIONS_PKG must be modified. Specifically, the control structures that are dependent on these enumeration literals must be changed to reflect the newly defined types.

*In the CSDL real-time testbed the package TL_OBJ_OPERATIONS_PKG is found in the file TL_OBJ_OPS_SP_B.ADA in the case of the single-processor version, and in the file TL_OBJ_OPS_B.ADA in the case of the multi-processor version.*

For the multi-processor environment, as explained above, the variable attribute **PROCESSOR** identifies the processor in which the variable resides. The processor identifiers are defined by the enumeration type PROC_TYPE, as illustrated in the following example:

```
type PROC_TYPE is (ENV_PROC, FSW_PROC);
```

New processor types may be defined by the user by modifying PROC_TYPE in package TL_VAR_TYPES.

**WARNING:** TIMELINER is very dependent on the literals defined by the enumeration type PROC_TYPE. If new enumeration literals are added to this list, or if the names of the enumeration literals are changed, a major update of the code in package TL_OBJ_OPERATIONS_PKG is required. Specifically, the control structures that are dependent on these enumeration literals must be changed to reflect the newly defined literals.

Further information about how TIMELINER uses the variable lists to "grab", "load", and "display" variables is given below in Appendix C.

When TIMELINER is incorporated into a multi-processor architecture, a separate variable list must be maintained for each processor forming part of the system. In this case the following attribute must be added to each entry in a variable list:

**PROCESSOR** An identifier that names the processor in which the variable resides. The user will define, as explained below, the valid choices for this identifier. *(In the case of the CSDL real-time testbed the available processor identifiers are ENV_PROC and FSW_PROC.)*

In a multi-processor environment, the variable list for each processor is sized by a separate parameter. Each list size parameter must correspond to the number of variables in each respective list.

Here is an example of a variable list entry for variable EVENT, which in this case is a 2x2x2 array of 32-bit integer type:

```
    ("EVENT                                        ",        -- Variable name
     OF_INT32,                                 -- Variable value type
     NUMBER_TYPE,                              -- TIMELINER type
     3,                                        -- Number of dimensions
     2,                                        -- Size of first dimension
     2,                                        -- Size of second dimension
     2,                                        -- Size of third dimension
     EVENT'ADDRESS,                            -- Variable address
     FSW_PROC),                                -- Processor where variable
resides
```

For use during execution time, the information in the individual variable lists is transferred to a composite variable list called TL_LOCAL_VAR_LIST that lies within the processor where TIMELINER operates. The parameter that sizes the composite list must be equal to the sum of the sizes of the individual lists resident in each processor.

*In the multiple-processor version of the real-time testbed, the variable list ENV_LIST for the "environment" processor is located in the file ENV_PROC_LIST_S.ADA and the variable list FSW_LIST for the "flight software" processor is located in the file FSW_PROC_LIST_S.ADA. The composite list TL_LOCAL_VAR_LIST is located in the package TL_VAR_TYPES found in the file TL_VAR_TYPES_S.ADA. In this case the composite list size LIST_SIZE must equal the sum of ENV_LIST_SIZE and FSW_LIST_SIZE.*

As stated above, the user may wish to add new variable types to the ones already available, as listed above under the variable list attribute VAR_VAL_TYPE. Such additional types, may be defined by the user in package TL_VAR_TYPES.

For example, if an 8-bit boolean is required by the implementation, it would be defined by the statements

```
type BOOL8 is new BOOLEAN;
for BOOL8'SIZE use 8;
```

Note that Appendix F of the appropriate Compiler User's Guide should be consulted to find the representation of the various base types available to the user for the particular compiler being used.

and the next control statement. The user was allowed to include other control statements within such a block by means of the DO statement and the END statement. A set of statements framed by a DO/END pair was considered as a block. This convention is similar to that used by the HAL language.

The Ada version of TIMELINER makes nesting of constructs more explicit by requiring that each "construct" begun by a WHEN, WHENEVER, EVERY, or IF statement be concluded by an explicit END statement. This method makes the way in which constructs are nested more explicit and frees the user from having to keep in mind the distinction between control and action statements.

*BEFORE/WITHIN instead of UNTIL/FOR:*

The earlier versions of TIMELINER used an UNTIL clause to terminate the operation of a WHEN, WHENEVER, EVERY or WAIT construct upon the occurrence of some other condition. The Ada version of TIMELINER uses the word BEFORE instead of UNTIL. The reason for this change is to improve readability in the presence of an OTHERWISE clause, as illustrated by the following English sentence, in which using "until" instead of "before" is awkward:

> When the pot boils before the timer buzzes, turn off the heat, otherwise reset the timer.

The earlier versions used a FOR clause to terminate the operation of a WHEN, WHENEVER, EVERY or WAIT construct upon the elapsure of a time interval. The Ada version uses the word WITHIN instead of FOR. The reason for this change is to avoid confusion caused by the entirely different meaning of FOR in the User Interface Language, and to improve readability. A FOR capability similar to UIL's may later be added to TIMELINER.

The reason for the change from FOR to WITHIN is illustrated by the following English sentence in which using "for" instead of "within" is awkward:

> When the light changes within 60 seconds drive on, otherwise honk the horn and drive on.

For users versed in the HAL or Fortran versions of TIMELINER, the meaning of BEFORE and WITHIN is identical to the meaning of UNTIL and FOR.

As described below the WAIT statement is handled differently in this version of TIMELINER. BEFORE/WITHIN cannot be used to modify the simple WAIT statement.

*Addition of an OTHERWISE clause:*

An optional OTHERWISE clause was added to the WHEN construct to allow the execution of TIMELINER statements upon the termination of the WHEN construct caused by fulfillment of a BEFORE or WITHIN clause.

## Appendix A. CHANGES IMPLEMENTED IN ADA-LANGUAGE VERSION

The version of TIMELINER written for and in the Ada language and documented in this User's Guide differs in some ways from the versions previously implemented in HAL and Fortran.

This appendix briefly describes these changes. Users without knowledge of the earlier versions may wish to skip this material.

The changes were motivated by (1) a desire to move closer to the conventions of the NASA Space Station User Interface Language, so that TIMELINER experience becomes more relevant to the UIL design, (2) a desire to make the conventions of the TIMELINER language closer to those of the Ada language, and (3) the need for additional capability.

The most significant changes between the HAL and Fortran versions and the Ada version of TIMELINER are the following:

*Separation of compile-time and run-time operation:*

In the earlier versions of TIMELINER the compilation of TIMELINER scripts occurred at the start of each run. In the Ada-language version, as described in Section 1.3 of this User's Guide, the compile-time and run-time functions are separated and the interface between them is in the form of an ASCII file.

In the earlier versions the compilation software formed part of the run-time link and scripts had to be recompiled each time even if there were no changes. This situation was acceptable for batch runs but less suited for the real-time test-bed application for which the Ada version was created.

*The "Bundle"*

The bundle is defined as a grouping of sequences and subsequences that form a whole, either because of some common purpose, or because they are targeted for use in a particular operating environment. For example, the "bundle" may be used to group a set of sequences and subsequences that are intended for execution in a particular processor of a distributed system.

The "bundle" level was not present in previous versions of TIMELINER, which were processed in a simplex environment. The "bundle" was added to the Ada-language version to facilitate the application of TIMELINER to distributed data management systems.

*No more DO/END pairs:*

The earlier versions of TIMELINER depended heavily upon the distinction between "control" statements and "action" statements. For example, the block of actions to be performed upon satisfaction of a WHEN statement was defined as the action statements lying between the WHEN statement

An OTHERWISE clause is illegal if the construct has no BEFORE or WITHIN clause and optional if it does.

The use of OTHERWISE as part of a WHEN construct allows different actions to be taken depending upon which of two conditions arises first.

*Change to the status of WAIT:*

In the earlier versions of TIMELINER UNTIL and FOR statements could be used in connection with WAIT, just as with WHEN, WHENEVER, and EVERY. Since explicit END statements are required for closing constructs in the Ada-language version, users of WAIT would be forced to include a meaningless END statement with a simple WAIT.

In the Ada language version WAIT functions more like an action statement. In other words, WAIT introduces a simple wait, cannot be modified by a BEFORE or WITHIN clause, and requires no END statement.

No capability is lost. For example the functionality provided by

```
WAIT <time_interval> UNTIL <condition>
```

can be provided by

```
WHEN <condition> WITHIN <time_interval>
```

when accompanied by an OTHERWISE clause. The capability provided by

```
WAIT <time_interval> FOR <time_interval>
```

was seldom if ever used and is probably meaningless.

## Appendix B. EXTENDED EXAMPLE OF A TIMELINER SCRIPT

This appendix contains an example of an actual TIMELINER script. This script is one used to control an end-to-end simulation of an AFE mission. The script was translated from the HAL/Fortran version of the language to the Ada version.

The sample TIMELINER script follows:

```
-- AFE FULL END TO END SIM

BUNDLE AFE_END_TO_END_SIM

-----------------------------------------------
-- OVERRIDES OF DEFAULT VALUES IN LOAD PROGRAMS
-----------------------------------------------

-- TIME AT START OF SIM IS MAY 13, 1994, 4 HRS. 47 MINS. 35.423 SECS.

    SEQ EPOCH

        LOAD CALN_UTC_AT_EPOCH 1994 5 13 4 47 35.423
        LOAD T_GMT      11508455.423
        LOAD T_SIM_REF 11508455.423

    CLOSE SEQ

    SEQ RUN_TIMES

        LOAD T_SIM  0
        LOAD T_STOP 15000

    CLOSE SEQ

    SEQ SIM_LOAD

        LOAD EDITOR_CNT 125
        LOAD STS_STATE_ACT ON

    CLOSE SEQ
```

*The preceding sequences will all execute on the first TIMELINER pass and then will be deactivated. The script writer has separated the time-zero loads into separate sequences for the sake of visibility.*

```
    SEQ ENV_LOAD

        LOAD AERO_MODEL  0
        LOAD ATMOS_MODEL 0
        WHEN EVENT_209 = ON
            LOAD AERO_MODEL      1
            LOAD ATMOS_MODEL     3
            LOAD STS_ATMOS_MODEL 0
        END WHEN
        WHEN MM_300 = ON
            LOAD EDITOR_PRINT_CNT 125
        END WHEN

        WHEN MM_400 = ON
            LOAD EDITOR_PRINT_CNT 2500
        END WHEN

        WHEN MM_500 = ON
            LOAD EDITOR_PRINT_CNT 125
        END WHEN

        WHEN MM_400 = ON
            LOAD EDITOR_PRINT_CNT 2500
        END WHEN

        WHEN MM_500 = ON
            LOAD EDITOR_PRINT_CNT 125
        END WHEN

        WHEN MM_400 = ON
            LOAD EDITOR_PRINT_CNT 2500
        END WHEN

        WHEN MM_500 = ON
            LOAD EDITOR_PRINT_CNT 125
        END WHEN

        WHEN MM_400 = ON
            LOAD EDITOR_PRINT_CNT 2500
        END WHEN

        WHEN EVENT_407 = ON
            WAIT 100
            LOAD UPLINK_CODE 102
        END WHEN

        WHEN EVENT_605 = ON
            WAIT 100
            LOAD UPLINK_CODE 103
        END WHEN

    CLOSE SEQ

    SEQ END_SIM

        WHEN EVENT_710 = ON
            WAIT 100
            LOAD T_STOP FROM T_SIM
            LOAD EDITOR_CNT 1
            LOAD EDITOR_PRINT_CNT 1
        END WHEN

    CLOSE SEQ

CLOSE BUNDLE
```

```
        WHEN EVENT_301 = ON
            LOAD  MASS_BIAS 3.4809
        END WHEN
        WHEN MM_400 = ON
            WAIT 200
            LOAD AERO_MODEL  0
            LOAD ATMOS_MODEL 0
        END WHEN

    CLOSE SEQ

    SEQ ILOAD

        WHEN EVENT_209 = ON
            LOAD ROLL_ACCEL_NOM  5.2
            LOAD PITCH_ACCEL_NOM 4.1
            LOAD YAW_ACCEL_NOM   5.1
        END WHEN

    CLOSE SEQ


----------------------------
-- MISSION TIMELINE CONTROL
----------------------------

    SEQ EVNT_CNTL

        LOAD MM_000_PRO ON
        WAIT 2
        WHEN EVENT_007 = ON
            LOAD UPLINK_CODE 101
            LOAD STS_SEP_DV  0
            LOAD STS_SEP_CMD ON
        END WHEN

        WAIT 00:30
        LOAD STS_SEP_DV  1
        LOAD STS_SEP_CMD ON

        WAIT 20:30
        LOAD STS_SEP_DV  7
        LOAD STS_SEP_CMD ON

    CLOSE SEQ

    SEQ PRINT_CONTROL

        WHEN MM_000 = ON
            LOAD EDITOR_PRINT_CNT 2500
        END WHEN

        WHEN MM_100 = ON
            LOAD EDITOR_PRINT_CNT 2500
        END WHEN

        WHEN MM_200 = ON
            LOAD EDITOR_PRINT_CNT 125
        END WHEN
```

The "grab" and "display" functions require that information (variable contents) flow from the FSW and ENV processors to the TIMELINER processor, where it is used in a decision or printed.

The "load" function requires that information compiled by TIMELINER be sent to the FSW or ENV processor to be inserted into a named variable.

In all cases the information to be transferred is buffered. It is then sent to the appropriate processor at a time specified by TIMELINER.

The following tables enable the transfer of variable data to/from the ENV and FSW processors:

**ENV_INPUT_VAR_TBL** The table containing the source and destination addresses of each of the ENV variable components to be transferred to the TL processor.

**ENV_LOAD_VAR_TBL** The table containing the source and destination addresses of each of the variable components to be transferred to the ENV processor.

**FSW_INPUT_VAR_TBL** The table containing the source and destination addresses of each of the FSW variable components to be transferred to the TL processor.

**FSW_LOAD_VAR_TBL** The table containing the source and destination addresses of each of the variable components to be transferred to the FSW processor.

The number of single variable components which are transferred to/from the ENV and FSW processors must be specified by the user.

**ENV_INPUT_TRANS** The number of single variable components transferred from the ENV processor.

**ENV_LOAD_TRANS** The number of single variable components transferred to the ENV processor.

**FSW_INPUT_TRANS** The number of single variable components transferred from the FSW processor.

**FSW_LOAD_TRANS** The number of single variable components transferred to the FSW processor.

The following example demonstrates the manner in which ENV_INPUT_TRANS and FSW_INPUT_TRANS are set:

Let's assume that our TL variable list contains four variables *(LIST_SIZE = 4)*, one of which is an ENV variable (*ENV_LIST_SIZE = 1*), and three of which are FSW variables (*FSW_LIST_SIZE = 3*).

Let's also assume that our variables are arrayed as follows:

## Appendix C. HOW TIMELINER ACCESSES VARIABLES

This appendix describes the method used by TIMELINER to access the simulation variables that are operated upon by a TIMELINER script. How the variables are made known to TIMELINER is described above in Section 8.2. This appendix explains how TIMELINER uses the variable list information described in that section.

The Ada-language version of TIMELINER may be incorporated into a single-machine "functional" simulation, or it may be incorporated into a multi-processor system.

In each case the upper level TIMELINER processing is the same. However, the process of accessing variables, whether to grab, load, or display them, is quite different. This appendix describes the mechanization adopted in the case of a multi-processor system, specifically the CSDL real-time test-bed for the Space Station DMS.

The real-time test-bed consists of three main processors:

386 (TL) The processor in which TIMELINER runs.

386 (FSW) The processor in which the "flight software" being simulated runs, just as it would run in a System Data Processor (SDP) forming part of the Space Station DMS.

486 (ENV) The processor in which the "environment" function, which closes the loops between the effector outputs and the sensor inputs of the flight software, runs.

These processors are linked together by a Multibus-2 interface. This interface plays the role of the actual on-board interfaces (1553 busses) that link Space Station SDP's to the MDM-controllers that in turn control spacecraft sensors and effectors. Delays are inserted as necessary to model the actual performance of the on-board system.

The Multibus-2 interface is also used to link the TIMELINER 386 processor to the other two machines. Since TIMELINER is a simulation tool, not part of the flight system, this interface does not require artificial delays to model real system performance.

The Multibus-2 interface is the avenue used by TIMELINER to access the variables that are subject to the following TIMELINER functions:

GRAB_VAR Obtains the value of an (unarrayed) variable for use by a control statement in making a decision.

LOAD_VAR Loads the variable (which may be arrayed) using data supplied by the user.

DISPLAY_VAR Prints (or otherwise displays) the value of a variable (which may be arrayed).

These three functions involve a two-way flow of information:

The user would also set

**FSW_INPUT_TRANS = 25**

noting that there are 3 FSW variables, two of which are arrays containing 24 individual variable components between them.

All user supplied information is manipulated by editing the package **TL_VAR_TYPES**. For the single-processor environment TL_VAR_TYPES is found in file TL_VAR_TYPES_SP_S.ADA. For the multi-processor environment the package is located in file TL_VAR_TYPES_S.ADA.

| | |
|---|---|
| EVENT | 3-dimensional array containing 8 elements |
| ROLL_ACCEL_NOM | un-arrayed variable |
| RVI_HAHPIWOF | 1-dimnsional array containing 6 elements |
| T_STOP | 2-dimensional array containing 16 elements |

The TL variable list, in this case, would be:

```
TL_LOCAL_VAR_LIST :=

(

        ("EVENT                                        ",
         OF_INT32,
         NUMBER_TYPE,
         3,
         2,
         2,
         2,
         EVENT'ADDRESS,
         FSW_PROC),

        ("ROLL_ACCEL_NOM                               ",
         OF_BOOLEAN,
         LOGICAL_TYPE,
         0,
         0,
         0,
         0,
         ROLL_ACCEL_NOM'ADDRESS,
         FSW_PROC),

        ("RVI_HAHPIWOF                                 ",
         OF_SCALAR_SINGLE,
         NUMBER_TYPE,
         1,
         6,
         0,
         0,
         RVI_HAHPIWOF'ADDRESS,
         ENV_PROC),

        ("T_STOP                                       ",
         OF_SCALAR_DOUBLE,
         NUMBER_TYPE,
         2,
         4,
         4,
         0,
         T_STOP'ADDRESS,
         FSW_PROC)

);
```

Given this information, the user would set

**ENV_INPUT_TRANS = 6**

noting that there is 1 ENV variable, an array containing 6 individual variable components.

The Charles Stark Draper Laboratory, Inc.

555 Technology Square, Cambridge, Massachusetts 02139 Telephone (617) 258-

SSF-LII-90-80

TO: Distribution
FROM: Don Eyles
DATE: June 29, 1990
SUBJECT: How TIMELINER Works

## Introduction

This memo describes the implementation of the TIMELINER language. TIMELINER is a simulation-input language widely used at the Draper Lab.

In the course of commenting on the space station user interface language (UIL) I have frequently mentioned the TIMELINER language. TIMELINER is relevant to UIL because TIMELINER is a language whose purpose is quite similar to that of UIL when UIL is functioning in the "compiled" mode as a language for specifying actions that should occur in the future under prespecified preconditions.

My comments on the space station UIL are in the following memos:

"Comments on UIL Spec", SSF-LII-90-63, May 14, 1990.

"Report on UIL Meeting in Reston", SSF-LII-90-77, June 21, 1990.

Having created and lived with TIMELINER for a number of years, its implementation seems somewhat obvious to me, and it may also appear obvious to readers. On the other hand, TIMELINER was developed without reference to any of the languages that are cited in the UIL Specification as having influenced the design of the UIL. Therefore TIMELINER offers a genuinely independent perspective, based on a proven system, from which UIL may potentially profit.

TIMELINER was developed about nine years ago. It has become quite popular at the Draper Laboratory, in part because it is easy to use. TIMELINER has been used to implement quite elaborate scripts, on the order of 100 sequences with a total length of over 2000 statements.

It is not the function of this memo to describe the TIMELINER language itself. The TIMELINER language is briefly summarized in an appendix. A more detailed description, in the form of a User's Guide, is available from the author upon request.

Versions of TIMELINER currently exist in both the HAL and Fortran languages. This memo contains estimates of the memory required by these implementations. This information may shed some light on the memory requirements for UIL. An Ada language version is contemplated for use in a real-time simulation of the space station DMS that we are starting to build at CSDL.

This memo concludes with a brief discussion of the changes that should be incorporated into the new Ada-language version of TIMELINER. To minimize risk this implementation will start with TIMELINER rather than UIL. However, the DMS-like system context of the simulation will require changes that will tend to drive TIMELINER closer to UIL. Therefore, with luck, we will be able to draw further lessons relevant to UIL.

**TIMELINER's Context**

The user of TIMELINER is the engineer wishing to conduct a test of some software system, typically relating to GN&C. The user employs TIMELINER:

- To control the simulation.

- To provide inputs such as data loads and astronaut keystrokes that must occur at certain times and under certain conditions during the simulation, including initial loads.

- To provide debug capabilities such as the ability to print a chosen variable, at a chosen frequency, without having to recompile and relink the software to include a print statement.

TIMELINER statement types provide mechanisms for organizing scripts (blocking statements), constructs for defining the preconditions for actions (control statements), and statements to perform actions (action statements).

TIMELINER processing divides into "initialization-time" and "execution-time" processing. What I call "initialization-time" corresponds to the "compile-time" processing of UIL scripts. It is the time when scripts are converted to executable form. In the case of TIMELINER this processing occurs when a simulation is initialized. In the case of UIL this processing would occur on the ground well in advance of the time when a script would be sent to the space station for use.

TIMELINER functions in connection with "batch" runs. However, TIMELINER is structured so as to allow additional statements to be input during execution-time for immediate processing. That is, there is no structural reason that the initialization-time processing cannot be performed during the run if additional statements are provided by some source.

**Serial/Parallel Capability**

The TIMELINER input for a run is called a "script". A script is made up of one or more (usually many) "sequences" and "subsequences".

To give the user of TIMELINER the flexibility he needs, TIMELINER provides for a "serial/parallel" organization of sequences. This means that an individual sequence executes serially, while separate sequences execute in parallel with each other.

The user uses the parallelism that exists among sequences when it is necessary to react to events and conditions that may occur in an unknown order. Each parallel sequence may be as small as a single "whenever" construct that will carry out certain actions whenever a specified condition occurs. Parallel execution means that each sequence maintains its own pointers and internal variables such that the "serial" progression through each sequence proceeds independently from other sequences.

The user uses the serial progression within a sequence when he desires that actions take place in a sequential manner, such as in the case of a sequencer that controls activities throughout a mission. A typical TIMELINER script contains a long sequence that provides mission-level sequencing functions, and a variety of smaller sequences designed to react to specific situations.

All sequences start out active. If a sequence is intended to begin at some point during the simulation this is implemented by incorporating, at the top of the sequence, a statement that will hold up further execution until the starting conditions, such as a time, come to pass. A sequence becomes inactive only when it has finished. TIMELINER sequences do not have the ability to turn each other on and off -- although this might be a useful capability to add.

TIMELINER is not interrupt driven. That is, it functions periodically, usually at the highest rate available in the simulation. On each iteration, TIMELINER processes the active sequences in the order that they occurred in the input script.

Besides sequences, TIMELINER provides for the creation of "subsequences". Unlike a sequence, a subsequence does not create its own independent stream of execution but forms part of the stream of execution created by the sequence that "calls" it. A subsequence may be called from different sequences, perhaps simultaneously. Re-entrancy is not a problem because TIMELINER input is a script that is interpreted, not a computer program, and the pointers and internal variables that control sequential execution within the script are local to the sequence that establishes each stream of execution. Subsequence calls may be nested.

The "subsequence" capability is regarded as useful by TIMELINER users. It allows them to create canned sequences to carry out various tasks (such as crew inputs needed to control a "burn"), which can then be called by the sequence that provides mission sequencing. Thus the upper level sequence stays cleaner and easier to modify.

## TIMELINER Statements

Within each sequence, a TIMELINER script is made up of "statements". There is no hierarchical level corresponding to "step" in the UIL language.

Three distinct types of statement exist in TIMELINER: blocking statements, control statements, and action statements:

Blocking statements are those that are used to organize statements into sequences and subsequences. TIMELINER blocking statements include SEQUENCE and SUBSEQUENCE header statements, and a CLOSE statement to terminate the input stream.

Control statements are those that control flow through the sequence. TIMELINER control statements consist first of the WHEN, WHENEVER, WAIT, EVERY, UNTIL, FOR, and IF/ELSE statements that define the conditions under which the sequence will proceed. These statement types are also called "condition" statements. Other control statements are CALL (used for invoking a subsequence), and the DO and END statements that are used as a pair, when necessary, to bracket blocks of statements that include other control statements.

Action statements are those that perform actions immediately when they are encountered. TIMELINER action statements include LOAD, PRINT, EXECUTE, KEY, and ABORT. In TIMELINER a group of action statements is always regarded as a block. Grouping by means of DO/END pairs is only required when control statements are included in a series of statements meant to be regarded as a block.

Note that in keeping with modern practice, my earlier memo "Comments on the UIL Spec" accepts the use of explicit "end" statements in conjunction with any condition statement that may form a block, replacing TIMELINER's practice of using DO/END statements only when necessary.

TIMELINER input is read in line by line. The type of a TIMELINER statement is always determined by the first word on a line. The end of a statement is normally marked by the end of the line. In some respects this simplifies the TIMELINER language, because no explicit keyword or semi-colon is needed to mark the end of a statement. Because of the identity between "statements" and "lines" the two terms are sometimes used interchangeably.

The only TIMELINER statement that is not confined to one line is the LOAD statement, when it is being used to set an array variable that requires more data than will fit on one line. In this case TIMELINER continues to read lines until the required amount of data is found. If a keyword indicating the start of a new statement is encountered before the LOAD data is complete, a cuss is issued.

An appendix to this memo contains a more detailed description of the TIMELINER statement types.

In the area of "action" statements, UIL needs constructs that go way beyond those provided by TIMELINER in order to deal with the diverse classes of object that exist in the space station system. In the area of "control" statements for specifying the preconditions for actions, I believe that TIMELINER's capabilities are superior to those provided by the version of UIL described in the current specification document.

## Error-Checking

The TIMELINER script for a given run is part of the simulation initialization, sometimes still called the "rundeck". The first function that must be accomplished at initialization-time is checking for errors. The purpose of error-checking is to help the user to create a script fulfilling his needs, and to insure that the run-time processing will not encounter problems.

Error-checking has two main aspects: language checking and object identification, as described below:

*Language checking...*

Language checking consists of determining whether the input statements conform to the rules of the language. Most of this can be done on a statement by statement basis. For example, does a WHEN statement consist, as it should, of the word WHEN and up to two relational expressions linked by AND, OR, or XOR?

Checking the syntax of the more complicated types of statement is normally done by comparing the "tokenized" format of the statement to a list of acceptable forms. Once the format is identified, the role of each "word" on the line is known.

Some syntax checking, such as insuring that an END statement exists for every DO statement, occurs on a sequence by sequence basis and the error message, if any, is printed at the conclusion of the sequence.

Other error checking, such as ensuring that every subsequence called by a CALL statement is present somewhere in the script, is performed at the end of the input stream.

*Object identification...*

The second part of error checking, object identification, has to do with the objects that provide the subject matter of TIMELINER statements. Such objects, called "nouns" in TIMELINER parlance, are mainly literals and variables. Other objects include executable software modules and keystrokes.

Objects must be identified. That is, the existence of objects named in TIMELINER statements must be verified. TIMELINER objects "exist" if they evaluate properly as a literal, or if they appear in the variable list available to TIMELINER -- and therefore in the subroutines that act on the objects. How objects are put in these lists is discussed later. Of course to be in the list they must first be present in the "flight" or "environment" software of the simulation.

If an object is named that is not present in the lists, an error message is generated. Once the object has been identified an error message can also be generated if, for example, the data provided for "loading" a variable does not fit the variable, or if a relational expression attempts to compare incompatible variables.

In evaluating compatibility, the benefit of the doubt is sometimes given. For example the user is allowed to compare a numeric variable or literal to a character string, such as one derived from a specified portion of a simulated CRT display. At execution-time such a comparison will always fail unless the character string in question converts properly to a number by the rules of the programming language in use.

If erroneous language or unknown objects are detected, TIMELINER attempts to generate error messages that are as informative as possible. As with most languages a single error may cause more than one error message to be issued, but it is usually clear where the real error is.

Even a single error is enough to prevent the run from occurring. However, error checking always continues through the entire script so that the user is able to eliminate all errors at once.

## TIMELINER Tabulation

Besides checking for errors, TIMELINER's initialization-time processing consists of reducing scripts to numerical form and entering the numbers into tables to provide data for the run-time execution of the script.

Thus, for TIMELINER, the role of an "intermediate language" is performed by numerical data in tables. The TIMELINER software that runs at execution-time knows how to use this data to get the job done.

In the case of UIL the "initialization-time" processing is sometimes referred to as "compile-time" processing. This term may be somewhat misleading because the intermediate form into which UIL scripts are converted is not necessarily in the form of executable object code. It may be a tabulated format like TIMELINER's.

In the case of TIMELINER, the use of a truly "compiled" intermediate language would defeat the intended purpose of TIMELINER in providing a means to change the operation of a run without forcing the user to recompile or relink any software. An intermediate language that took the form of computer language would probably also require more storage space that do the numeric tables that are used.

Two main sets of tables exist, the first containing data relating to each sequence, the second containing data relating to statements.

Sequence tables exist for such information as the character-string form of sequence names, whether each is a sequence or subsequence, and the number of the first and last line of each sequence or subsequence -- used for referencing each sequence's portion of the statement tables. Other tables exist for the pointers and other variables used in the run-time processing of each sequence.

In the statement tables, an entry exists for each statement. Since there is only one statement table for all sequences the length of each sequence is unconstrained. Only the total number of statements is limited by the size of the tables. Of course, table sizes may be increased if required by the application.

The statement table identifies the type of statement numerically, and contains other data related to the statement. A statement table entry may contain pointers that point to entries in other tables. For example, the entry for a WAIT statement may contain a pointer to a "noun" table that describes the type of noun (variable or literal) specifying the time interval in question. The "noun" table contains either the "address" of the variable or a further pointer to a table containing the literal.

The table organization is tailored to the need to reduce the computation time required by the run-time processing. In most cases the table entries become "indices" (or subscripts) in the run-time software.

Memory utilization was not considered to be a great problem when TIMELINER was originally implemented. Thus, for example, the statement table contains 10 numbers for each statement, even though many types of statement use only one or two. Further hierarchization of the TIMELINER tables could improve memory usage considerably.

TIMELINER initialization-time processing is performed during <u>one</u> pass through the script.

## <u>Execution-Time Processing</u>

The execution-time (or "run-time") processing performed by TIMELINER consists of processing in turn, in the order that they appear, each sequence that has not become inactive by completing its statements.

Execution-time processing uses only the data tabulated in sequence and statement tables and in subsidiary tables pointed to by them. At run-time, TIMELINER scripts no longer exist in character form. As described in the next section, TIMELINER reconstructs the original statements into character form for display.

It is very unlikely that a given sequence will "do something" on every pass. On most passes, a sequence will be waiting for the precondition to exist that will permit its next action to occur. Since TIMELINER is not "interrupt-driven" it must "poll" each sequence on every pass. For TIMELINER to be efficient, the time taken by each such poll must be minimized.

Therefore, TIMELINER is designed such that on typical passes only two "if" questions must be asked about each sequence: <u>Is it active</u>? Does the condition <u>being waited for exist yet</u>? Even if many sequences are in operation, executing two "if" statements for each does not require an excessive amount of time.

## TIMELINER Printing

TIMELINER "prints" information for use by the user. Of course this information may be directed to a data set or to a terminal display instead of to a printer.

Printing occurs both at initialization-time and at execution-time:

*Initialization-time printing...*

Initialization-time printing consists of an echoing of the script provided by the user, to which error messages are added where necessary. Additional information provided as part of the initialization-time printing is a statement number for each statement, and indentation.

Statement numbering is continuous across sequence boundaries. Statement numbers provided at initialization-time provide a reference that is sometimes useful in interpreting the run-time printing.

The user may begin each statement at any column position. When printed at run time indentation is used to convey the blocking of the script created by the constructs used. For example, a block of action statements executed upon fulfillment of a WHEN statement condition is indented with respect to both the WHEN statement and the condition statement that follows the actions.

TIMELINER initialization-time printing includes a table usage summary so the user will know if he should increase table sizes to accommodate larger scripts. Increasing table size requires recompilation of the TIMELINER programs, but is rarely necessary once suitable table sizes have been set.

*Execution-time printing...*

Execution-time printing consists of discrete packages. Each package represents the statements executed by a given sequence, on a given pass. If multiple sequences advance on a given pass, separate print packages are generated for each. Each package begins with a header line that gives the name of the sequence and the time. Within a package, each statement bears its statement number.

Printing occurs only when a sequence advances. Thus, a WHEN statement is printed only when the WHEN condition is fulfilled. A WAIT statement is printed only when the time interval expires. Sometimes a terminating condition, such as that specified by an UNTIL clause, causes a statement such as WHEN or WAIT to be terminated and the consequent action statements skipped. In such a case only the UNTIL line is printed. Similarly, in the case of an IF/ELSE, if the ELSE block is executed rather than the IF block, only the ELSE block is printed.

This practice was adopted to reduce the confusion that might be caused by printing at run-time a statement containing a condition that was not fulfilled. However, in some cases the user may need to use the statement number provided to refer back to the print-out generated at initialization time in order to see the context of the printed statements. If output were terminal directed, colors might be used to allow the context provided by non-executed statements to be shown in a non-confusing fashion.

As described earlier, at initialization time the statements making up a script are "reduced" to numeric form and stored in tables. Statements are not saved in their character-string form. Accordingly, the run-time printing uses the stored information to reconstruct each statement for printing.

Statement reconstruction at run-time, besides saving memory, guarantees that the statement is printed as it was tabulated and executed. This practice provides a cross-check to insure that the tabulated form of each statement corresponds to the statement that was input by the user. TIMELINER is fully checked out and does not make errors of this type. Nevertheless this practice gives users a warm feeling.

Alternate forms are allowed for some TIMELINER reserved words. The run-time printing uses standard forms. Thus, whether the user originally wrote "AND" or "&", it will always appear as "AND" in the run-time printing.

## How TIMELINER Deals with "Objects"

This subsection discusses how "objects" are made known to TIMELINER, and how TIMELINER scripts then get at the objects during processing.

In the existing implementations of TIMELINER, the main objects involved are computer variables that are present in the simulation. This includes both the variables used by the "target" flight code, and the variables used by the "environment" models.

Four functions are required with respect to each variable:

FIND This function, which occurs at initialization-time, consists of determining whether the named object exists, and providing an "address" to be used during execution-time by the GRAB, LOAD, and PRINT functions.

GRAB Obtain the value of the variable for use in condition statements. This function has a single, unarrayed value as its output, but subscripts allow any member of an array to be referenced.

LOAD Load the variable (which may be arrayed) from data supplied as part of the sequence, and then print it.

PRINT Print the variable in a tidy format.

TIMELINER must interpret a script that names the "nouns" or "objects" in question. Thus TIMELINER must make the connection between a character string naming an object and the object itself.

The process of going from character string to the variable itself is not facilitated by the HAL or Fortran languages, or by any other language known to me. For applications such as TIMELINER or UIL, this would be a useful feature. It could be implemented by retaining, at run-time, the compile-time tables that link the character string form of a variable to the variable's address.

Since such tables are not available, TIMELINER must provide its own -- and it must do so in a fashion that allows users to add and take away variables in a convenient way. In the HAL and Fortran language versions two different strategies are employed:

* In the HAL version, users add variables once, in character form, to a common area, along with information describing each variable. Then, within separately compiled GRAB, LOAD, and PRINT subroutines the HAL macro capability is used to expand the variable description into language statements to do the required action. In the common area itself, the statements are used to create an array that allows variables to be "found" and their "addresses" determined. In other words, the same original statement is interpreted by different macros in each situation.

* Fortran does not provide "macro" capability matching that of the HAL language. Therefore in the Fortran case another strategy is used. An off-line program is provided that has the capability to "read" files containing the common areas in which variables are found and to generate a Fortran subroutine with multiple entry points to provide the FIND, GRAB, LOAD, and PRINT functions for the variables. This subroutine is then compiled and added to the simulation.

In both cases, the code that actually performs the GRAB, LOAD, and PRINT functions is in the form of a large, hierarchic DO CASE or computed GOTO statement. The "address" provided by the FIND function is in fact a set of indexes used at execution-time to reach the proper "case".

Besides variables, the main "objects" of interest to TIMELINER are executable subroutines. In the existing implementations the capability of executing such subroutines has not been widely used. Therefore the entry of new executable subroutines is done manually. This feature could be automated in a way analogous to the way variables are dealt with.

Other objects of interest to some versions of TIMELINER are keystrokes. Within any application the keyboard is unlikely to change very often. Therefore a table of keys is maintained manually.

UIL must deal with a much wider variety of objects than TIMELINER. On the other hand, since UIL is embodied in a computer program, dealing with any object can probably be thought of as a combination of setting variables and invoking a subroutine that, using the information in the variable, performs the processing and I/O necessary to do the action. Therefore, while the parsing of action statements at initialization-time will be more difficult, and a large number of subroutines will be needed, the execution of such statements during run-time may not be much more complex than in the TIMELINER case.

**How much memory does TIMELINER need?**

The job of estimating the memory required to implement a piece of software is notoriously difficult, and conservative practice often causes estimators to apply large "fudge factors" to avoid underestimation.

Because it may be useful in helping to estimate memory requirements for UIL, this section cites the size of the existing TIMELINER implementation. The numbers fall into three categories:

* The memory required by the TIMELINER "language processor" -- the portions of the TIMELINER code that implement the language. The size of the TIMELINER language processing software is independent of the number of "objects" dealt with, but would increase if additional language constructs were included.

* The memory used by the TIMELINER "common area" that includes most of TIMELINER's own variables. The predominant component of this category is the tables into which processed TIMELINER statements are placed for run-time execution. This category corresponds to storage requirements for the "intermediate language" version of scripts. The memory required for the tables depends upon how they are sized, i.e. upon how many sequences, lines, nouns, literals, and so forth, are allowed. This number is also affected by the number of "objects" known to TIMELINER. The common area also contains tables used in the execution-time processing of scripts.

* The memory required for the mechanisms that "get at" TIMELINER's "objects" -- mostly variables. These mechanisms consist of subroutines to "grab", "load", and "print" variables, as described above. Variables range from single booleans, integers and scalars, up to large arrays and (in the HAL case) structures that include components of varying type. The memory required for these functions depends upon the number of variables that must be dealt with.

As stated earlier, TIMELINER exists in two versions, in the HAL and Fortran languages. The memory sizes for both versions are summarized below.

The HAL language figures are taken from the most elaborate TIMELINER implementation in use at CSDL, that used in our OFS8A Shuttle simulation.

The Fortran language figures are taken from the CSDL simulation of the Aerobrake Flight Experiment. Some features are not identical to the HAL language version.

**HAL language, TIMELINER memory requirements in bytes:**

| function | bytes |
|---|---|
| language processor (1969 HAL statements) | 51633 |
| *breaks down as follows (based on statement counts):* | |
| *initialization-time processing (39%)* | *20137* |
| *execution-time processing (45%)* | *23235* |
| *non-essential code in compilation unit (16%)* | *8261* |
| language processor internal variables (most of which are used in initialization-time processing) | 17992 |
| compool (including tables for "intermediate language") sized for 512 sequences, 10192 statements | 632256 |
| subroutine to "grab" 2321 variables | 61012 |
| subroutine to "print" 2321 variables | 193969 |
| subroutine to "load" 2321 variables | 118170 |

**Fortran language, TIMELINER memory requirements in bytes:**

| function | bytes |
|---|---|
| language processor | 37363 |
| *breaks down as follows:* | |
| *initialization-time processing* | *19860* |
| *execution-time processing* | *17503* |
| compool (including tables for "intermediate language") sized for 128 sequences, 4096 statements | 500284 |
| subroutine to provide "find", "grab", "load" and "print" functions for 1045 variables | 151598 |

The differences between the HAL and Fortran versions have to do with minor changes in the capabilities present in each. For example, the Fortran version does not contain logic to check the existence of subsequences called using the CALL statement. Another difference lies in the fact that the HAL language implementation must deal with a wider variety of variable types than the Fortran implementation. In addition, the efficiencies of the two languages are unequal.

The "compool" memory estimates, which relate to intermediate language storage requirements, are somewhat misleading because tables are not sized in an optimal way. For example, the Fortran version allows for up to 1024 character string literals of 64 characters each (65536 bytes) although it is very unlikely that sequences totalling 4096 lines will include that much character-string data. More reliable intermediate language sizings will emerge from the upcoming Ada language implementation.

## Implications of TIMELINER Memory Estimates

The numbers given in the preceding section allow certain conclusions to be drawn that may be relevant to the implementation of UIL.

First, the figures given above suggest that the language processor function of a language such as UIL or TIMELINER is a relatively minor consumer of memory, as compared to other functions. Naturally this category will increase if further capabilities are added to the language. However, the addition of further language constructs probably does not contain a high risk of "blowing up" the memory required to implement such a language.

Second, the statistics suggest that the major consumers of memory for a language such as TIMELINER or UIL are (1) the subroutines required to deal with the "objects" that provide the subject matter of UIL/TIMELINER statements, and (2) the tables in which the executable form of the language is stored.

In the area of the subroutines required to deal with "objects" the space station UIL is particularly vulnerable. The wide diversity of objects, and the variety of the actions that may be commanded with respect to the objects, will require a large number of "back-end" subroutines to implement. Since the space station crew must have the ability to input UIL statements to act on <u>any</u> of the space station objects, the full family of such routines will have to be present in each MPAC. For UIL embodiments in various ground facilities, only a subset of such routines may be required.

The memory required for the storage of UIL scripts is presented in a pessimistic light by the statistics given above, because the TIMELINER implementation did not heavily emphasize the development of the optimum tabulation scheme. In addition, the memory required for scripts can be reduced, within <u>each</u> on-board processor, by distributing scripts among the various DMS processors, possibly including SDP's as well as MPAC's, for execution.

I believe that the tabulated form of "intermediate language" will be more compact than other alternatives such as storing UIL scripts in raw, character form, or in the form of a compiled computer language.

## Changes for the Ada-language version of TIMELINER

As mentioned above, an Ada-language version of TIMELINER will be built for use in the real-time space station DMS simulation that we will be building at the Draper Lab under our Level II contract.

This simulation urgently needs a version of TIMELINER to facilitate control of the simulation. For that reason the first version will be a relatively simple translation of the existing versions of TIMELINER into Ada.

However, since the point of this simulation is to create a system that meaningfully resembles the space station DMS, the Ada-language version of TIMELINER will naturally evolve in a direction that will increase its resemblance to UIL.

This section briefly describes some of the TIMELINER changes that will eventually be incorporated into this implementation:

*Add the concept of "environment"...*

The SSF UIL contains a concept called "environment" that is somewhat ill-defined at this time. The concept of "environment" is discussed in my earlier memos, cited in the introduction.

As I see it, the most valuable function of the environment is as a device for grouping together a set of "sequences" (called "procedures" in UIL parlance) that are related to each other by reason of their purpose or the discipline with which they deal.

I believe that a device for grouping procedures in required both for logistical and administrative reasons, and because characteristics of the execution environment in which procedures should execute are most appropriately specified at this level. The term "bundle" has been suggested as an alternative to the term "environment".

The following aspects of the operational environment for a related set of sequences would probably be specified at the "environment" or "bundle" level:

* The scope of the constituent procedures, i.e. the classes of object to which the procedures require "read" or "write" access.

* The particular processor on which the procedures are meant to be executed.

* The execution rate required by the procedures.

* The priority of the grouped procedures.

* Initial activation status of the procedures within the bundle.

In TIMELINER terms, the "environment" or "bundle" is an additional hierarchical level above the level of sequences and subsequences.

The Ada-language version of TIMELINER for our real-time DMS simulation should include the concept of "bundle", and the ability for the user to specify at least the target processor and the iteration rate applying to all the sequences in the bundle.

*Getting at variables in other processors...*

In existing implementations, TIMELINER executes as part of a simulation running on a mainframe-type computer. Thus, all objects with which TIMELINER may need to deal are immediately at hand -- once TIMELINER has been enabled to recognize them, as discussed above.

The Ada-language version of TIMELINER may at execution-time run in several separate processors, and each embodiment may require access to variables that are present in another processor. Means must be provided to create this access.

Two approaches suggest themselves:

(1) The TIMELINER execution-time processors in each machine will have the ability to deal with the variables in that machine only, but communications links will allow each TIMELINER embodiment to request action by any of the other TIMELINERs. That is, the GRAB, LOAD and PRINT functions for the variables in a particular processor may be invoked by the TIMELINER software in another processor.

(2) As in the case of space station "RODB services" a data base will exist where updated copies of all variables of interest to TIMELINER will be maintained. This data base might be called SODB (simulation operational data base) in the case of objects that do not pertain to the target DMS software. In this case TIMELINER will ask for information from other processors, and act on variables, by means of a combination of RODB and SODB services.

At any rate, for TIMELINER to be really useful, it should be able to obtain, and to act upon, variables that reside in another processor. I suspect that the first solution mentioned above will be better, but this area needs thought.

*Immediate-command capability...*

TIMELINER was originally structured to allow additional TIMELINER statements to be input <u>during</u> execution-time. In this case the initialization-time TIMELINER software is called again to analyze the input statements and add them to the executable tables. However, this capability has not been used by the existing TIMELINER implementations.

In the case of our DMS real-time simulation such immediate-action commands may have a useful role. During some types of run a man may be in-the-loop and may wish to modify some object or request that it be displayed or printed. It may be useful to be able to input in real time a sequence that uses a WHENEVER statement to capture data relating to a glitch that has been observed.

This capability, which corresponds to the immediate-input mode of the space station UIL, should be fully implemented in the new, Ada-language version of TIMELINER.

*Sequence-control capabilities...*

As stated earlier, all TIMELINER sequences start out in an active state. Internal constructs are used to determine when each should begin its major function.

The UIL language contemplates the use of explicit commands to turn "procedures" on and off and to temporarily suspend them. This capability is probably a useful one, because it allows an architecture in which a man-in-the-loop, or a "master" UIL procedure, may control the execution of other procedures. This capability may apply to individual procedures or to all the procedures in a given "environment" ("bundle").

These capabilities should be added to the new Ada-language version of TIMELINER.

*Syntax changes...*

Certain improvements in the constructs of the TIMELINER language should be incorporated, including:

* Constructs such as WHEN, WHENEVER, EVERY and WAIT should have an optional OTHERWISE clause that may contain actions to be executed when the construct is terminated because of another condition or the elapsure of a time interval.

* The clauses currently used by TIMELINER to specify such termination conditions for a WHEN, WHENEVER, WAIT or EVERY statement are UNTIL and FOR. Because they read better with OTHERWISE, these should be changed to BEFORE and WITHIN.

* Consideration should be given to switching to explicit END statements to conclude constructs initiated by statements such as WHEN, WHENEVER, WAIT, and EVERY instead of the present system of DO/END pairs.

* The IF/ELSE statement should have the capability of IF ELSE clauses. At present the "if else" function must be accomplished by increasing the level of nesting.

* Certain constructs available in UIL such as FOR and CHOICE would be useful additions to the TIMELINER language. There is no reason to add UIL constructs such as WHILE or VERIFY because existing constructs include these functions.

*Additional action statements...*

As the building of our simulation proceeds, types of action not now performed by TIMELINER may become desirable, such as, for example, actions connected with controlling equipment periferal to the simulation. The Ada-language version of TIMELINER should be structured to facilitate adding new types of action statement.

*A better tabulation scheme...*

By "tabulation scheme" I mean the format of the tables into which the "intermediate language" data upon which the execution-time TIMELINER processor will operate is placed at initialization time.

TIMELINER's tabulation scheme did not receive much thought when TIMELINER was first built because memory constraints were not considered important. Thus, for example, the main statement table contains 10 cells for each statement, even though only a few statement types need that many.

A more-optimum tabulation scheme will minimize the size of the "intermediate language" files that must be present in each of the processors in our simulation, and will allow more realistic estimates of the intermediate language storage requirements of UIL to be derived.

## APPENDIX:  TIMELINER CONSTRUCTS

This appendix contains a very brief description of the constructs of the TIMELINER language.

In this description, TIMELINER statements are grouped into three categories:

* "Blocking statements" that implement the division of a TIMELINER script into an arbitrary number of sequences, which are executed in parallel with each other.

* "Control statements" that are used to control the flow through each sequence, in accordance with specified conditions.

* "Action statements" that are used to specify actions that are to occur under conditions defined by the control statements.

*Blocking statements...*

SEQUENCE

The sequence header is of the form:

```
    SEQ <sequence name>
        OR
    SEQUENCE <sequence name>
```

where <sequence name> is any alphanumeric word without blanks. A sequence header terminates the previous sequence and starts a new sequence to be executed in parallel with other sequences.

SUBSEQUENCE

The subsequence header is of the form:

```
    SUBSEQ <subseq name>
        OR
    SUBSEQUENCE <subseq name>
```

where <subseq name> is any alphanumeric word without blanks. A subsequence header terminates the previous sequence and starts a new sequence that does not establish its own parallel stream of execution but instead is called by another sequence.

CLOSE

A CLOSE statement consists of the single word "CLOSE". It terminates the preceding sequence and closes the input stream.

*Control statements...*

WHEN

The WHEN statement is of the form:

```
    WHEN  <comparison>
        OR
    WHEN  <comparison>  AND|OR|XOR  <comparison>
```

where <comparison> is a relational expression involving literals or variables. When the stated condition is met the action statements that immediately follow the WHEN statement are executed, and then the sequence advances to the next condition statement.

WHENEVER

The WHENEVER statement is of the form:
WHENEVER <comparison>
*OR*
WHENEVER <comparison> AND|OR|XOR <comparison>
where <comparison> is a relational expression involving literals or variables. When the stated condition is met the action statements that immediately follow the WHENEVER statement are executed and then the sequence returns to the WHENEVER statement, which then waits for the next OFF-to-ON transition of the condition.

WAIT

The WAIT statement is of the form:
WAIT <numeric>
where <numeric> is a numeric variable or literal. The WAIT statement waits the specified time (reevaluated every pass if a variable) and then the action statements that follow are executed.

EVERY

The EVERY statement is of the form:
EVERY <numeric>
where <numeric> is a numeric variable or literal. The EVERY statement immediately executes the action statements that follow, waits the specified interval, and again executes the action statements, and so on.

UNTIL

The UNTIL statement is of the form:
UNTIL <comparison>
*OR*
UNTIL <comparison> AND|OR|XOR <comparison>
where <comparison> is a relational expression involving literals or variables. The UNTIL statement must immediately follow a WHEN, WHENEVER, WAIT or EVERY statement. If the UNTIL condition is fulfilled before the condition in the preceding statement, the action statements that immediately follow are skipped and execution begins with the next WHEN, WHENEVER, WAIT or EVERY statement. (In the UIL proposals in my previous memo, this function is renamed BEFORE.)

FOR

The FOR statement is of the form:
FOR <numeric>
where <numeric> is a numeric variable or literal. The FOR statement must immediately follow a WHEN, WHENEVER, WAIT or EVERY statement. If the FOR interval expires before the condition of the preceding statement is fulfilled, the action statements that immediately follow are skipped and execution

resumes with the next WHEN, WHENEVER, WAIT or EVERY statement. (In the UIL proposals in my previous memo, this function is renamed WITHIN.)

IF

The IF statement is of the form:
IF <comparison>
*OR*
IF <comparison> AND|OR|XOR <comparison>
where <comparison> is a relational expression involving literals or variables. If the specified condition is fulfilled the action statements that immediately follow are executed.

ELSE

The ELSE statement consists of the single word "ELSE". It must follow an IF statement and its action statements. The action statements following the ELSE line are executed instead of the action statements following the IF line if the IF condition is <u>not</u> fulfilled.

DO

The DO statement consists of the single word "DO". It is required (in a pair with the END statement) to bracket groups of statements that contain WHEN, WHENEVER, WAIT or EVERY statements and which are to be executed as a package, such as following fulfillment of an outer WHEN, WHENEVER, WAIT or EVERY statement.

END

The END statement consists of the single word "END". It closes the group of statements started by a DO statement.

CALL

The CALL statement is of the form:
CALL <subseq name>
It causes an immediate branch to the named subsequence. Following completion of the subsequence, control returns to the statement following the CALL statement. A called subsequence behaves like a package framed by DO and END statements.

*Action statements...*

ABORT

The ABORT statement consists of the single word "ABORT". When encountered it causes the simulation to be terminated.

LOAD

The LOAD statement is of the form:
LOAD <variable name> <data>
*OR*
LOAD <variable name> <scale factor> <data>
where <variable name> identifies any variable known to TIMELINER and <data> consists of numeric, boolean or string data commensurate with the variable. When the statement is

executed the data, modified in accordance with the optional <scale factor>, is loaded into the variable, and then the variable is printed.

LOAD FROM

The LOAD FROM statement is of the form:

```
    LOAD <variable name> FROM <variable name>
        OR
    LOAD <variable name> FROM <arithmetic combo>
        OR
    LOAD <variable name> FROM <function> OF <variable name>
```

The LOAD FROM statement allows a variable to be loaded from a variable, from a simple arithmetic combination of variables, or from a function of a variable. Arithmetic operators that may be used to form a "combo" are addition, subtraction, multiplication, division, exponentiation, and modulo. Allowed functions include square root, absolute value, rounding, and trig functions. The LOAD FROM statement allows elementary computations to be performed in the TIMELINER language. Complex algebra requires multiple steps using some variable as an "accumulator".

PRINT

The PRINT statement is of the form:

```
    PRINT  <variable name>
```

where <variable name> identifies any variable known to TIMELINER. When the statement is executed the variable is printed.

DUMP

The DUMP statement is of the form:

```
    DUMP  ALL
        OR
    DUMP  <common area name>
```

where <common area name> identifies a common area known to TIMELINER. When the statement is executed every variable in the common area (or in all common areas) is printed.

EXECUTE

The EXECUTE statement is of the form:

```
    EXECUTE  <subroutine name>
```

where <subroutine name> identifies any subroutine known to TIMELINER. When the statement is executed the named subroutine is executed.

KEY

The KEY statement is of the form:

```
    KEY  <series of keystrokes>
```

where <series of keystrokes> is a series of legal keystrokes relating to the flight system under simulation. When the statement is executed the keystrokes are input to the simulation as though they came from a keyboard

A more detailed description of the TIMELINER language, in the form of a User's Guide, is available from the author.

The Charles Stark Draper Laboratory, Inc.

555 Technology Square, Cambridge, Massachusetts 02139 Telephone (617) 258-

SSF-LII-90-77

TO: Distribution
FROM: Don Eyles
DATE: June 21, 1990
SUBJECT: Report on UIL Meeting at Reston

This is a report on my meeting June 12 in Reston with User Interface Language (UIL) guru Randy Davis and Peg Snyder of Level II.

Also in attendance were John Hinkle, Ralphine Ippolito, Cathy Cannon, and (for the first hour) Jim Duda. The purpose of the meeting was to discuss the issues and proposals I raised in my memo "Comments on UIL Specification" dated May 14, 1990.

There was a good exchange of views, consensus was reached on some issues, and a method for dealing with several important questions still needing resolution was worked out.

Most important, I think, a major question was brought to light as to the scope of the UIL language. Specifically, is UIL the language in which procedures that must react to conditions and events, such as OMA short term plans (STP's), will be written? Or is UIL only needed for executing "canned" sequences? A lot depends on the answer to this question.

Peg Snyder set the stage: The existing UIL Spec (2.1) will be baselined as it stands. Simultaneously, a block CR will be created to make any modifications that may come out of the UIL Implementers meeting, which is the current forum for dealing with UIL issues. Our purpose this day was for Randy and I to evolve a position on the issues I raised. She hopes that I will be satisfied by whetever position arises from this discussion, but if I do not I may submit a CR at a later time.

I began by stating "where I'm coming from":

I like lots of things about the present UIL; in fact I love the really powerful capabilities envisioned for action statements. I am not trying to "gold plate" the UIL. I am trying to offer UIL insights derived from my experience in building and using the TIMELINER language. TIMELINER is quite similar in purpose to the "compiled" mode of the UIL language. TIMELINER is a language proven over 9 years of extensive use at CSDL, with a well-understood implementation, and popular with users.

Finally, I made the point that a UIL with greater capabilities, such as I propose, is not necessarily more expensive (in money, CPU, or memory) than a less capable one, because a more capable UIL may be cleaner to implement.

Randy then gave us some background:

UIL has evolved from experience with the STOL, CSTOL and GOAL languages, and for that reason some of the UIL decisions had a "political" component. These languages are "action oriented", and thus they contribute more to the UIL's action statements than to the constructs for specifying when and under what conditions actions will occur -- as is evident from the relative immaturity of the latter in UIL. Fortunately, as Randy pointed out, the "first stake in the ground" for UIL implementers will be to deal with the "action statements", so there is a little breathing room for working out better condition constructs if that is necessary.

Randy described the UIL implementers group that is meeting monthly. The composition is mainly implementers, those who will build particular UIL embodiments at various facilities. Users, those who will write UIL scripts, are also represented. Lockheed has two people working full time on UIL. Also represented are IBM, Harris, and some others. I will attend at least the next session in July, for reasons apparent later.

Principal implementation issues currently being discussed include:

(1) The role of the UIL "environment" -- which Randy believes is evolving in a direction similar to that I suggest.

(2) The role of "event handlers" -- which depends on the degree to which UIL needs to react to events and conditions.

(3) Data types, object classes, etc. -- about which I have little to say.

At this point we break for lunch, planning to resume by going through my memo section by section.

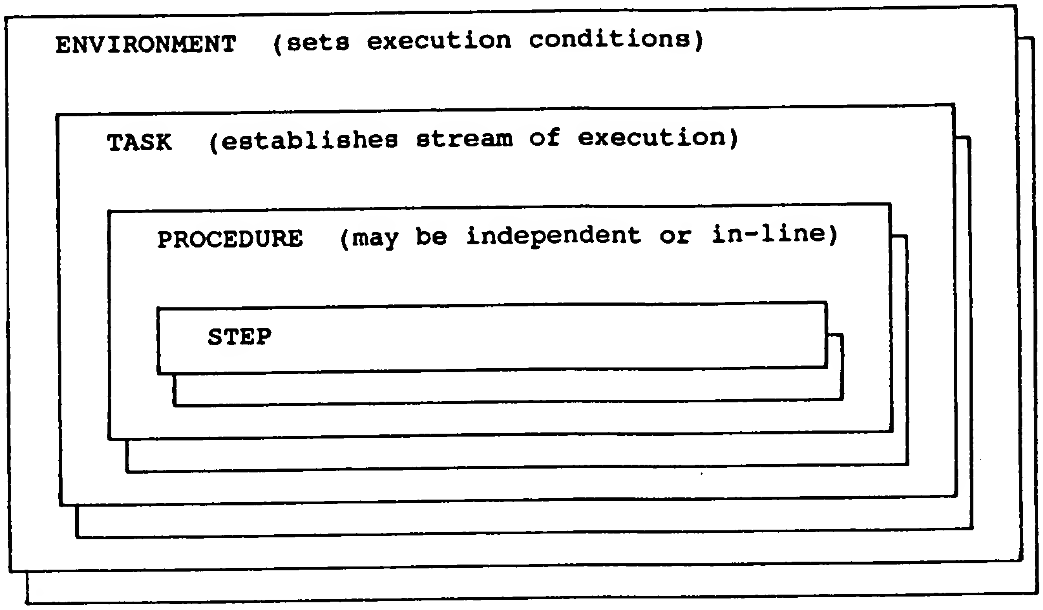
The following sections give a blow-by-blow account of our meeting. In some cases I have taken the opportunity to carry the discussion one step further, by adding comments and clarifications that occurred to me after the meeting.

## Section 1.1, on the role of "environment"...

Randy Davis began by saying that he agreed with much of the discussion in this section of my memo, and that he thought the position of the implementers group was moving in the same direction as my suggestions.

The present inclination of the implementers is to add an additional entity called the "task" to the UIL hierarchy as a device to create a parallel "string" of UIL execution. A task would include multiple procedures, executed sequentially.

Adding the entity "task" would deal with my objection to the use of "environment" in the UIL Spec merely as a device to create parallelism. Thus, the envisioned hierarchy is something like this:


ENVIRONMENT (sets execution conditions)
TASK (establishes stream of execution)
PROCEDURE (may be independent or in-line)
STEP

As seen by Randy, somewhat contradicting its treatment in the UIL Spec, an "environment" is not a thing but simply an operating environment in the sense of the English word. "Environment" is specifically not a "package" for the grouping of related procedures.

I had several problems with this hierarchy.

First, I thought that the most useful role for the "environment", as I had understood it, was as a packaging tool for the grouping of multiple, parallel procedures with a related purpose or discipline. In fact I thought "package" was a better name. Randy argued that "package" was unsuitable because it has a special Ada meaning that might create confusion, so at the suggestion of Ralphine Ippolito we started using the word "bundle".

Randy's comments revealed that "environment" was not being thought of as a grouping device at all. For example, he indicated in response to a question from John Hinkle that if UIL procedures dealing with three different disciplines but with more or less the same operating conditions were executing in one machine they would probably be in the same "environment", with aspects that varied, such as scope, specified at a lower level.

I argued that it was essential to have some tool for grouping related procedures, both for logistical/administrative reasons and because most aspects of operating environment -- such as machine, scope, priority, iteration frequency -- would tend to be common among a group of procedures for performing some particular purpose. Thus it would be easier to specify them at the level of a "bundle" of procedures than to have to repeat it for each procedure (or task).

Second, I thought that the proposed hierarchy already had too many levels, and this would be compounded if we added the concept needed for grouping procedures with the same environment but different purposes that we were calling "bundle". That would create five levels in all.

Third, I was not persuaded that the new entity called "task" should establish a new level of hierarchy. That is, rather than have tasks (each creating an execution stream), which in turn contain procedures, which may in turn call other procedures in-line, it would be better to rename the existing "procedure" as "task" in cases where it is being used to create a stream of execution, and to retain the name "procedure" only for entities that are called by tasks for in-line execution.

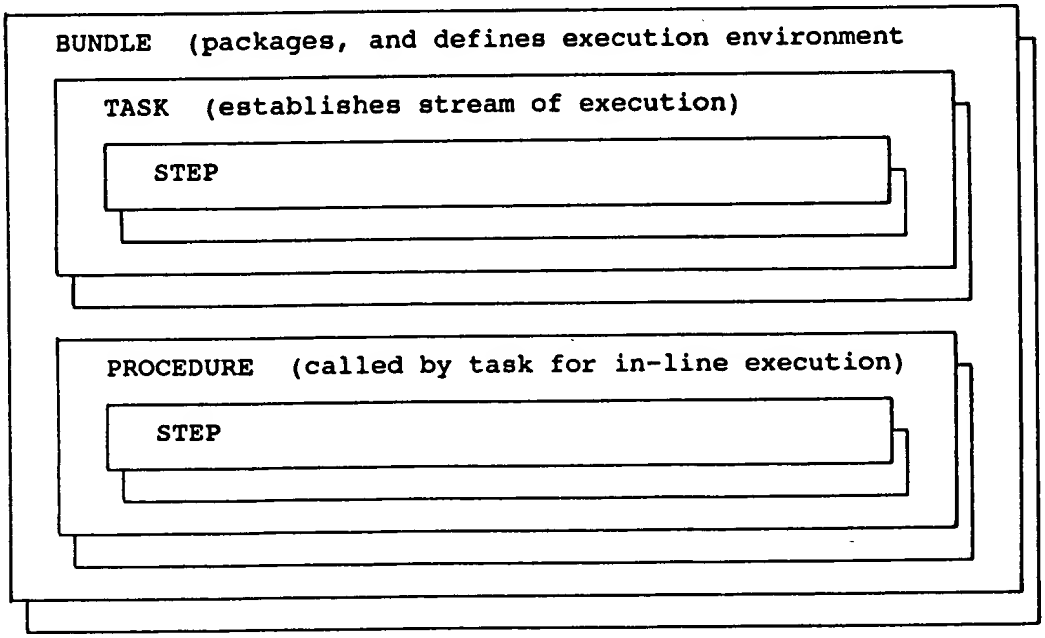
In this arrangement, although called by a "task", a "procedure" would not be <u>inside</u> a task. Rather, it would be another sort of animal within a "bundle". This is appropriate because the same "procedure" could conceivably be "called" by more than one of the "tasks" in a "bundle".

Apparently another reason for the creation of a new hierarchic level called the "task" is to create a single identifier, pertaining to each sequential string, so as not to confuse users by the changing identifiers of the procedures making up a given stream of execution. Although I didn't say this at the time, the identifier question could be solved by letting an identifier read something like this:

A/b/c/d...

Where "A" names the "task" that creates the string and "b" and "c" and so on represent procedures invoked (in-line) by the "call" function -- which may be nested to some number of levels.

The hierarchy created by my suggestions would look something like this:


BUNDLE (packages, and defines execution environment
TASK (establishes stream of execution)
STEP
PROCEDURE (called by task for in-line execution)
STEP

To my mind this in cleaner. It gets rid of the confusion as to what an "environment" is, it creates the "bundle" to serve the need for a packaging device, and it distinguishes in a visible way between entities that create a stream of execution and those that are simply called in-line. The hierarchy is reduced to three levels and four types of entity rather than the five types and levels that result if the needed concept of "bundle" is simply added to the existing hierarchy.

The names themselves aren't important. What are called "tasks" and "procedures" in the above picture I would really have preferred to call "sequences" and "subsequences" (see below). What is important, in my opinion, is that entities with different functions have different names. I especially dislike the idea that "procedure" be used <u>both</u> for entities in independent streams and those that are not. There may be a better word than "bundle" for the entity that groups "tasks" and "procedures", but it was evident from our conversation that using the word "environment" for this purpose may be confusing.

Randy was open-minded on this subject, and stated that the door had already been opened for discussion of such points by the UIL implementers group. He invited me to bring this issue to the next UIL implementers meeting, which will be on July 9-10 at Lockheed in Houston.

Under the heading of my section 1.1 there was also discussion of the "scoping" function:

In particular, should the user be required to scope procedures by specifying explicitly in advance, at the top of whatever level, the object classes that would be dealt with; or should the UIL "compiler" have the job of determining *ex post facto* the object classes that were dealt with and inserting this information where appropriate, presumably at the top of the executable file.

Randy wanted something at the top so that a procedure did not get three quarters of the way through and <u>then</u> discover that some object was unavailable. John seemed not to like the idea of "scoping" UIL sequences at all because he didn't want the intermediate language executor (ILE) at execution time to have to trace the ramifications of a UIL entity to discover what objects were being used and therefore their availability.

Randy argued that some sort of scoping was necessary to bar an experimenter, who is only interested in a few tens of objects, from having access to the full range of station objects -- so he cannot accidentally do damage outside of his area.

I pointed out that anything that any tracing ILE can do <u>at the top</u>, could be done at compile time. Randy said that object availability could change between compile time and execution time, but John or I pointed out that it could even change between the top of a UIL procedure and a point later in the procedure.

Anyway, I think there was a little cloudiness here between the need for scoping as a limiting device to keep that experimenter out of trouble and the need for scoping as a way to foresee whether the objects required for a piece of UIL are all available.

Another thing that came up under this heading was my argument that the current PERFORM statement, which can do either an in-line call (if no "within environment" clause) or start an independent stream (with "within"), was potentially confusing, and that it would be better to use a different word, such as CALL, for the in-line type of invocation. At any rate the PERFORM statement will need revision if "task" replaces "environment" as the device for establishing an independent stream.

Randy sorta agreed with this, but said that he thought the implementers would not like it. He also said that since PERFORM was an action statement, and not really part of the language, this change could be made later without impact if desired. This was a little too professorial for me, and I pointed out that it might technically be true, but that action statements dealing with language entities such as procedures were relevant to the language in a way that action statements dealing with controlling a valve, for example, were not.

My own preference, assuming a hierarchy similar to the one I proposed above, would be to define separate sets of action statements for starting or stopping whole "bundles", for dealing with "tasks" within that bundle, and finally a "call" statement for use in invoking "procedures" for in-line execution. I believe this would help the user to avoid confusion.

About this time John asked an interesting question: When UIL is waiting for some condition to occur, does it have to check for that condition on every pass?

Randy's answer, I believe, was "probably not."

My answer was, "probably yes", and that it didn't matter because asking one or two "if" questions for each active stream of UIL was not a burden.

John replied that what concerned him was not the time taken to ask the "if" questions, but rather the time and bus loading involved with obtaining the value of the object. He wanted UIL to be able to take advantage of the RODB services' capability of supplying a value only when the object in question changes.

This was a real good point, which I had not thought about. In fact, an "if" question may still be asked every pass for each UIL stream that is waiting for something, but the question may involve a flag indicating whether a relevant object/attribute's value has changed rather than the value itself. (This may be simpler than logic to deactivate a stream and reactivate it when a value changes.) Then, if it has changed, the condition can be reevaluated. On the other hand, it may be just as simple to check the existing (unupdated) value as it is to check a change-flag for the object. The important thing is that UIL be designed so that it does not need a value to be refreshed unless it has changed.

Section 1.2 about "procedure"...

Randy Davis differed strongly with my argument that "sequence" was a better name than "procedure". I had argued that "procedure" invited confusion with software procedures and with crew procedures, and that "sequence" better conveyed the serial nature of a UIL stream.

Randy began by saying that in plain English a "sequence" was a series of anything and a "procedure" was a series of instructions -- so "procedure" was more descriptive of a UIL entity.

Also, he pointed out that "procedure" is not used by <u>all</u> programming languages to denote a called entity. He argued further that "confusion" between "UIL procedures" and "crew procedures" was not necessarily a bad thing, since the two mapped together. Finally, he noted that usually the term "UIL procedure" was used, thus making the meaning clear.

Perceiving that name-changing was not viewed sympathetically by this audience I said I could live with the nomenclature "procedure".

The more interesting part of this discussion came when Randy cited the paragraph in my memo that reads:

> The use of the term "sequence" is also appropriate because it is the nature of UIL to provide an outer shell for the SSF DMS that provides the functions of a "master sequencer".

Randy did not agree with this statement. He made the very revealing observation that this was not the way UIL had been viewed during its design, and that this role would be played by OMA entities such as short term plans (STP's) and mission activity plans (MAP's). UIL would be used only for relatively "canned" sequences whose interactions with the "outside" were simple enough as to be within the capabilities of existing constructs such as "event handler".

Randy went on to say that this is why UIL did not provide an enhanced "level of reactivity" to external conditions, such as that provided by the new constructs I proposed in my memo. This is evidently what Randy had meant when he had characterised my memo as enlarging the "scope" of UIL.

Well, this did represent a major disconnect, because I had in fact assumed that UIL was the language in which procedures that must react to various events and situations, such as STP's, would be written, because what other language existed that was at all suitable for this purpose. Thus I had perceived the existing constructs such as the "event handler" for reacting to situations as inadequate.

This stemmed from my assumptions about where UIL would be used, which were as follows:

(1) To off-load the crew, since SSF is so complex, so the crew can do science, etc., instead of spending all their time operating the spacecraft.

(2) Therefore UIL will do jobs traditionally done by the crew.

(3) For such jobs the crew must retain visibility, so a language must be used that is comprehensible to crewmen, rather than a programming language such as Ada. Thus UIL may be thought of as a "visibility tool" aimed at telling crew and controllers what the spacecraft is doing, in a form they can readily understand and modify if necessary.

(4) OMA procedures such as STP's should be written in UIL if (as is the case) their activities need to be visible to the crew and controllers.

<u>In other words the criteria for the use of UIL is whether crew and ground controllers require visibility into the operation of the procedure in question.</u>

John Hinkle got involved here. If I got it right, he said that MAP's and DTL's (detailed time lines) would definitely be written in UIL, and that STP's probably would, although this decision had not been made.

I asked, if they don't use UIL, what language would they use?

I pointed out that all this had major program implications. Some language is needed for writing STP's, and the only options available now are UIL and Ada. Hardly anyone thinks Ada is suitable for this because of the readability issue. Thus, if UIL is not used for this purpose, a third language is necessary, and someone had better get started on it.

I stated than in my view, since UIL already contains most of the needed capability, it would be much better and cheaper to increment the capabilities of UIL than to create some third language.

Peg Snyder took these points. If this meeting accomplished nothing else than to reveal this disconnect I believe it was worthwhile.

(About this time John Hinkle had to leave to go to another meeting, and this prompted the following repartee between him and Peg. John said, "This is great, but it took us 2 1/2 hours to get here". Peg replied that he was lucky it hadn't taken 2 1/2 months. "When in this program have you ever gotten anywhere worthwhile in 2 1/2 hours?")

Randy acknowledged that UIL had been designed with rather rigid, sequential, "canned" tasks in mind such as long strings of *do this, wait, do that, wait,* and so on. I gather this stemmed from the background experience of the STOL, CSTOL and GOAL languages. This background explains UIL limitations such as the ones inherent in the existing "event handler" construct.

This may also help explain the fact that UIL, unlike TIMELINER, did not emphasize the capability of establishing multiple, parallel streams of sequential execution.

Randy indicated that if in fact UIL needed to be capable of doing the more complex and "reactive" sorts of thing that would be involved in such areas as OMA short term plans, then such constructs as the ones I proposed in my memo should be considered.

Section 1.3, on "event handlers"...

This issue overlaps with the preceeding discussion.

Randy Davis began by saying that the definition of "event handlers" has a "political" dimension. The existing concept of the "event handler" is taken from the GOAL language, and reflects the desire of the community experienced with GOAL to have their familiar friend on board.

Randy defined events as "specialized notifications equivalent to interrupts". In a ground test environment such things may be more prevalent than on-board the station.

I believe Randy conceded my point that the "event handler" treats as a special case one specialized class of object (events), and cannot deal with multiple conditions, or with non-events such as a voltage drop or a data spike.

He understood my proposal, which he described as creating WHEN and WHENEVER statements into which would be "subsumed" such existing functions as the "event handler" and the VERIFY construct. The question in his mind was, "do the requirements warrant that level of reactivity?"

This got us back into the question of whether UIL is a language for such things as STP's. However, I would be willing to argue that even regardless of that question, for UIL to go as far as it does in terms of "reactivity", but not go the last little distance, would be faulty design. For example, leaving STP's aside, a scientist writing a UIL procedure to control his experiment will surely be frustrated if UIL does not facilitate reacting in real-time to a spike in the data.

At any rate, Randy stated that the "event handler" is not one of the language constructs that anyone plans to implement anytime soon, so the "door has been opened" for further consideration of this matter. This is another issue that I am invited to discuss with the UIL Implementers group in July.

Meanwhile, Peg assigned John Hinkle to try to get an answer to the question of whether UIL will be used for the STP's, etc. Evidently OMA will work off data tables, so this question boils down to: what language tool will be used in creating the OMA data tables? I gather that the question is on the table already, that UIL is a candidate (other candidates, if any, unknown to me), and that the decision is due by August or September.

(I have heard talk that a more sophisticated "rules-based" language may be needed for writing STP's. I hope not, because there may be a swamp on that road. However, I think a case could be made that constructs such as the (proposed) WHENEVER construct, used in parallel, would allow a sort of "rules-based" system to be implemented using UIL.)

Section 1.4, on "step" and the GOTO statement...

Randy stated that "step" was also a political issue because it gave people a "warm fuzzy" feeling, and was good for monitoring the progress of procedures and for mapping UIL procedures to documents defining crew procedures.

He also maintained that GOTO's tend to be used judiciously, that GOTO is present even in structured languages, and that removing GOTO would be perceived as a user-unfriendly act. In fact, he even cited Dykstra (though I forget exactly how) in support of having GOTO in the UIL.

I was persuaded, mostly by the argument that "step" was useful for monitoring purposes.

Section 1.5, on "command qualifiers"...

Randy conceded that there were ambiguities connected with "command qualifiers". However, they were not meant to be used to define the times at which actions should occur, but rather to pass time information downwards to the particular object that would execute the command, which may "have a life of its own".

I accepted this. Randy agreed that the text needed clarification and that the words such as AFTER, BEFORE, AT, UNTIL and so forth should be looked at with a view to changes to avoid confusion.

Section 1.7 and 1.8, on VERIFY and CHOICE...

This really had to do with the motivation of not having the IF/ELSE, the eschewal of which I had lamented because of its advantages as a clear, well-understood concept.

Randy Davis gave a little history, of which the following is probably an over-simplification: IF/ELSE used to be in the language, but since it was redundant with VERIFY (when without a WITHIN clause), and with CHOICE, the decision was made to eliminate one of them. Since by then the VERIFY/WITHIN had come into being (for another purpose), the decision was to get rid of IF/ELSE instead of VERIFY.

Randy also stated that another problem with IF/ELSE is that if there is a string of ELSE IF's, the state of objects could change between the various questions such that none of the alternatives would pass. CHOICE solves this because the data is grabbed at the top.

I responded that IF/ELSE could solve this by acquiring all the data needed at the top of the whole construct, and Randy agreed this could be done. Anyway, I do not see why UIL would spread the evaluation of an IF/ELSE construct over more than one pass.

(A more serious problem of this type may exist. To evaluate any condition expression, UIL may require data from diverse sources. At any particular time, due to bus delay uncertainties, this data may not be a time-homogeneous set. This needs thought.)

Incidentally, my complaint on page 19 that CHOICE did not permit nesting was in error.

I suppose I should shut up about it, but I have to admit that in my mind the case against IF/ELSE is not completely closed. For example, consider the situation in which a cascade of "if" checks must be made in which the data for each "if" involves different objects.

In this case the simple CHOICE statement is not enough, because CHOICE is designed for checking various states of a single object. A nesting of VERIFY or CHOICE statements would be required. With each nesting the indentation becomes deeper and the statement becomes harder to read. In contrast, the IF/ELSE statement would keep all the alternatives at the same level -- which to my way of thinking is more in accord with the logical content of the statement. The following comparison indicates what I mean:

```
Verify X = 1                              If X = 1 Then
   Then                                      A
      A                                   Else If Y = 2 Then
   Otherwise                                 B
      Verify Y = 2                        Else If Z = 3 Then
         Then                                C
            B                             End If
         Otherwise
            Verify Z = 3
               Then
                  C
            End Verify
      End Verify
End Verify
```

Which alternative is more civilized?

The problem of data changing during evaluation of the expression is no argument in this case because both possibilities suffer equally from it. And CHOICE is no better than VERIFY in this instance.

Personally, I would like to keep CHOICE, restore IF/ELSE, and delete VERIFY. IF/ELSE would replace VERIFY without WITHIN, and the proposed, more general WHEN/WITHIN would replace VERIFY/WITHIN.

Section 1.9, on REPEAT...

My objections to REPEAT had to do with the fact that it did not permit a time interval to be specified for the repetition, that variations in its placement were confusing, and that it was sometimes superfluous.

Randy Davis's first defense of REPEAT was that it is right out of Ada. I don't know if this is a persuasive argument where UIL is concerned, since UIL is not designed for people who know Ada. Otherwise we could do without UIL and use Ada.

He pointed out that REPEAT in connection with the WHILE and FOR statements provides a "token" to mark the end of the content of the WHILE or FOR construct. This was a good point.

Randy did not object to my argument that UIL is fundamentally different that Ada in the sense that it is designed to operate <u>over</u> <u>time</u>, and that therefore a REPEAT that causes maximum rate repetition is unsuitable for UIL. He acknowledged that REPEAT would not be implemented to loop within a single UIL iteration anyway, because if it did nothing else would get done.

Randy seemed open to the idea of adding an EVERY statement, whose time interval could be set to the UIL iteration rate to reproduce the present REPEAT, or to a longer interval if the user sees fit. In this case the EVERY statement would replace the stand-alone usage of REPEAT but not its use in connection with WHILE and FOR. This would have the added benefit of avoiding the confusion caused by the varying placement of the reserved word REPEAT.

<u>Section 1.10, on WHILE...</u>

I criticised WHILE because it is redundant with the use of EXIT IF inside a REPEAT loop, and because it may imply that WHILE can be used to determine the conditions under which some action starts, not just (as is the case) when it should stop.

I suggested that my proposed replacement of WHILE by EVERY/BEFORE was cleaner and clearer, and I believe Randy was willing to consider getting rid of WHILE if my set of constructs were adopted for other reasons. And I would be willing to keep WHILE if it were not.

<u>Section 1.11 and 1.12, on FOR and EXIT...</u>

We didn't really argue these points.

I like FOR, and I am persuaded by Randy's "token" argument that the REPEAT reserved word is needed to assist parsing.

I stated my argument about EXIT but I don't remember Randy's response. Neither of us considered it a major point.

### Section 1.13, on the UIL Spec document itself...

Randy Davis defended the organization of the document, which starts with the details and moves upward towards more general considerations, on the grounds that this is how language specifications are customarily written. He acknowledged that a better overview should be provided as introductory material.

### Chapter 2.0, on my proposed new constructs...

Randy had not read this part of my document in detail, and we did not discuss it very much. I made general comments to the effect that my goal was to define an integral set of constructs that would replace the existing medley of constructs (listed in section 2.2) for controlling when and under what conditions actions should occur.

I cited the following chart (from page 35 of my memo) to illustrate my contention that a more capable design may sometimes be less expensive than a less capable one:

| | logical expression | time interval |
|---|---|---|
| one-time | WHEN | WAIT |
| establish a loop | WHENEVER | EVERY |
| terminate main construct and execute optional OTHERWISE block | BEFORE | WITHIN |

This chart shows the proposed WHEN, WAIT, WHENEVER, EVERY, BEFORE and WITHIN constructs in a grid. WHEN, WHENEVER, and BEFORE have commonality because all take logical expressions. WAIT, EVERY and WITHIN have commonality because all take time intervals. In the other dimension, commonalities exist between WHEN and WAIT, between WHENEVER and EVERY, and between BEFORE and WITHIN.

An implementation of UIL can take advantage of such common characteristics to simplify its processing.

Incidentally, since UIL will be independent of the concept of a "line" and will not use a statement terminator such as a semi-colon, a reserved word such as THEN should be added to the proposed syntax for the proposed WHEN, WHENEVER, EVERY and WAIT statements to mark the end of the specified condition.

Randy's summary:

Randy Davis made the following summary of agreements and disagreements:

> He agrees that "environments" and "event handlers" need further consideration, and that this consideration should be done in the context of the existing UIL Implementers group.
>
> He strongly disagrees that "procedures" should be named "sequences".
>
> He disagrees that PERFORM (independent execution) and CALL (in-line execution) need to be distinguished from each other, although it would be easy to do.
>
> He agrees that my WHEN and WHENEVER constructs are a viable alternative to the current concept of "event handlers", and that this is another item for the UIL implementers group.

My Summary:

This meeting provided historical background -- including the fact that UIL is rooted in action-oriented test-input languages such as GOAL -- that helps explain why UIL is strong in the area of action statements, and somewhat less strong in the area of constructs for controlling actions taking place, over time, in the future.

Therefore I believe I contributed to the UIL effort by bringing into the picture the background of the TIMELINER language, which has complementary strengths and weaknesses.

Peg Snyder made it very clear that UIL will not be changed unless it requires change to fulfill its required purposes. Therefore, the chief driver for UIL improvement is the question of how and where UIL will be used. For example, is UIL the language in which such procedures as OMA short term plans will be written? (And if not UIL, what?)

If the answer to that question is "yes", then Randy and I seem to agree that more capable constructs for controlling future actions may be required, and that my proposals form a viable basis for such changes.

A forum exists for the further discussion of such topics, i.e. the UIL Implementers group, to which the scene now shifts.

The Charles Stark Draper Laboratory, Inc.

555 Technology Square, Cambridge, Massachusetts 02139 Telephone (617) 258-

SSF-LII-90-63

TO: Distribution
FROM: Don Eyles
DATE: May 14, 1990
SUBJECT: Comments on UIL Specification (Revised)

## INTRODUCTION

This memo contains my comments on the SSF User Interface Language Specification (USE 1001 Version 2.1 dated March 15, 1990).

This memo revises an earlier version dated April 20, 1990. Changes include an update to reflect version 2.1 of the UIL Spec, minor changes in the new constructs that are proposed, and reorganization of the memo into separate parts dealing with specific criticisms (Part 1) and with the proposed new constructs (Part 2). All sections contain some changes.

* * * *

The UIL Specification seems strong in some areas. In other areas I believe there is room for improvement.

The strength, in my opinion, is in the discussion of objects and attributes -- the hierarchical definition of the "nouns" and "adjectives" that constitute the subject matter of UIL statements -- and in the capability of the "command" or "action" statements that will operate upon these objects.

The weakness is in the means proposed for the organization of command statements into "UIL procedures" meant to accomplish some purpose over the course of <u>time</u>.

The organizational elements proposed in the UIL Spec are somewhat ill-defined and contradictory. Although concurrency is possible in certain cases, the UIL does not appear to support a general capability to organize a given function into an array of sequences that execute in parallel.

In the area of constructs for specifying the preconditions for actions, the UIL Spec offers a collection of special cases rather than a general system. Some functions that can be stated simply in English require awkward constructions to carry out. This memo proposes an alternative set of "primitive" constructs that I believe supplies the requisite functionality more cleanly.

Most of my comments apply only to the "compiled" version of the UIL language used to control actions in the future. The immediate-command mode, in which the sequencing function is provided by the crewman or controller who is entering the commands, concerns only the "action" or "command" statements that I believe are the strong point of the UIL defined in the document under review.

I was happy to read, on page 1-11, that "UIL should be designed to be easy to learn and remember" and that "UIL should be more similar to natural spoken English than to a traditional programming language", because this is the premise underlying many of my comments. The quoted statements reinforce my belief that the creation of a straightforward user interface language is more than a "personal preference" -- it is a program necessity.

## ORGANIZATION OF THIS REVIEW

This review contains two major parts.

Part 1 contains specific comments aimed at particular sections of the UIL Specification. First, the hierarchical organizational elements such as "environment" and "procedure" are discussed. Next the specific statement constructs are considered. Recommendations for change are made that in some cases refer downwards to the new language constructs proposed in Part 2.

Part 2 defines several new constructs that should be considered for inclusion in the User Interface Language. It begins with a short section describing a CSDL test-input language called TIMELINER that is relevant to UIL. In the next sections new constructs are introduced that enhance the ability of a UIL sequence to specify, in a straightforward way, the preconditions that must exist for a particular action to be performed. Comparisons are made between how functions would be written using the proposed constructs and how they would be written using the existing ones.

Throughout this review specific modifications are suggested. Unless someone talks me out of it, it is my intention to submit these recommendations to the appropriate design review as a RID.

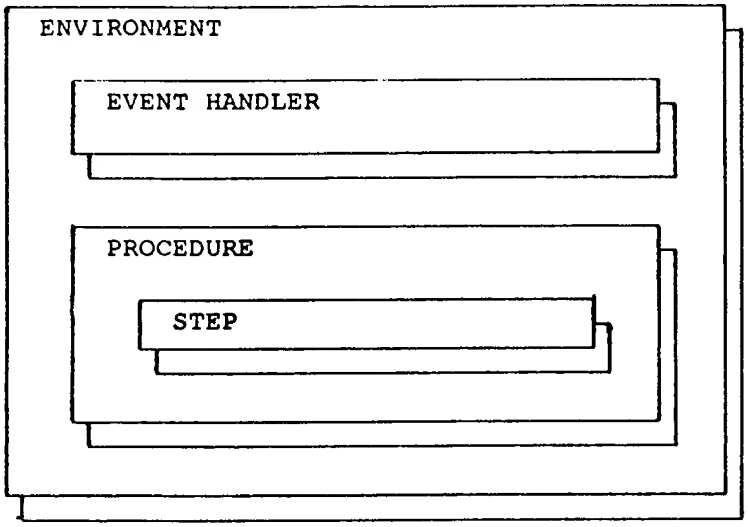
## 1.0 SPECIFIC COMMENTS

This part contains comments on specific sections of the UIL Specification. The order is not necessarily the same as the order in the UIL Spec because it seems more logical to address architectural issues before details.

In some cases I give examples of how particular functions would be written using new constructs proposed later in this memo. These examples are largely self-explanatory -- as any UIL sequence ought to be -- but the reader may wish to refer ahead to Part 2 for detailed descriptions of the new constructs.

### 1.1 Comments on "Environment" (Sections 3.6, 4.16)

The document defines a hierarchical organization for UIL entities that, as best I can tell, is as follows:



The "environment" is the outer level of the UIL hierarchy. Its main purpose appears to be to "determine which object classes and which objects are available in the execution environment".

In other words, the environment is a "scoping" device, intended for specifying the "scope" available to its constituent parts. As such, the concept of environment is a very powerful and useful tool. It would be the level at which UIL sequences designed for some particular function, and therefore with common scope, would be packaged. This purpose alone is sufficient to justify the existence of "environments".

It is obviously intended that different environments will execute concurrently, in parallel with each other. This is appropriate because different UIL environments will tend to be dedicated to different purposes that it would be difficult to integrate into a single stream of execution.

Unfortunately, the UIL Spec also uses the "environment" for other purposes, with the result that its usefulness as a scoping and packaging tool is compromised.

Because a UIL environment will execute in parallel with other ones, it is apparently assumed that the environment is the only device that can be used for this purpose. This assumption is illustrated by the description of the PERFORM statement (in section 3.7) where it is stated that the PERFORM statement will execute the invoked procedure concurrently only if a WITHIN clause is used to place it in its own environment.

Elsewhere it is implied that each environment will contain only one active procedure at a time. The description of the SUSPEND statement (4.16.4.2) states that SUSPEND "shall suspend indefinitely the procedure or event handler active within the environment". This is contradicted on the same page by the description of the TERMINATE statement (4.16.4.1), which states that TERMINATE "shall terminate all procedures and event handlers currently active within the environment". The intention behind these statements is unclear.

I strongly believe that the writer of a UIL function needs the flexibility to be able to create multiple active procedures within a given "environment" package. Each "procedure" would execute sequentially, but separate procedures would execute concurrently with each other. Thus a serial/parallel architecture would exist within each environment, as well as among environments.

This is necessary because the purpose fulfilled by a given UIL package may require multiple parallel sequences to be carried out. Forcing users to write each "purpose" in a single stream of execution may compromise understandability and may make it difficult to handle situations in which two or more independent conditions must be watched for, especially if they may occur in any order.

The UIL Spec recognizes that a given purpose may require concurrent "event handlers" that work in cooperation with procedures. What I am suggesting is that this concept should be generalized to allow multiple concurrent procedures as well.

The multiple concurrent procedures making up an environment would be "bound" to each other, in the sense that the term is used in section 1.5.3.2.2 of the UIL Spec, precisely by being placed in the same environment. If the environment is defined as a collection of procedures related to a particular purpose, then it is a suitable tool for both the "scoping" and the "binding" of its constituent sequences.

Note that "concurrency" or "parallelism" does not mean true simultaneous operation in the sense of computers running in parallel. What is meant is that independent streams of operation are maintained for each sequence. Within a given environment all the sequences will probably be processed in the same computer. On a given iteration the sequences would be processed in the same order as they appear within the environment, and thus a sort of priority among sequences would be established by their placement.

Allowing concurrency among the procedures within an environment will end the need to create separate environments, which may have the same purpose and the same scope, merely in order to create concurrency. This will avoid the weakening of the concept of "environment" as a tool for grouping and scoping procedures that have related purposes, and will comply with the principle of the "separation of concerns". In other words:

ENVIRONMENT <==> scoping and grouping of related PROCEDURES

PROCEDURE (SEQUENCE) <==> establishing concurrency within ENVIRONMENT

RECOMMENDATION: Make it clear that the concept of "environment" is intended for the purpose of scoping and packaging UIL "procedures" (or "sequences") that have related purposes, and that an "environment" may simultaneously contain multiple "procedures" executing in parallel.

If this recommendation is adopted statements such as PERFORM, TERMINATE, SUSPEND and RESUME will become useful both for operating on whole environments and for operating on procedures within environments. From outside one may wish to perform, terminate, suspend, or resume a whole environment. Within an environment one may wish to perform, terminate, suspend or resume individual procedures.

The PERFORM statement as it is currently defined is potentially misleading because it functions in fundamentally different ways depending on whether a subsidiary clause is added. PERFORM creates a concurrent stream if a WITHIN clause is used to name a separate environment for the procedure being invoked, but executes the procedure "in-line" if the WITHIN clause is absent. Visibility would be considerably enhanced if two different statements were defined to perform the two types of "call". Therefore I suggest that a CALL statement be added to create an in-line call, and that the PERFORM statement be reserved for initiating a concurrent sequence.

RECOMMENDATION: Modify the PERFORM statement such that it always creates a separate concurrent stream of execution. Modify the PERFORM, TERMINATE, SUSPEND and RESUME statements so that they may be used to initiate, kill, suspend or resume an individual "procedure" within an "environment" as well as a whole "environment". Add a CALL statement to the UIL for invoking a "sub-sequence" for in-line execution.

Although the next points may be more germane to the UIL implementation than to the language itself, I want to point out that the "environment" is probably the level at which it is appropriate to apply a priority and an iteration frequency to UIL sequences.

It would be better to apply a priority to a UIL package, assigned according to the package's purpose, than to the individual sequences within the package, because those sequences will tend to be designed to work together as an integral whole. In situations where not all UIL functions can be performed it is probably better to do load-shedding on an environment by environment basis, as priority dictates, rather than by trying to "micro-manage" their constituent parts.

The iteration rate is the frequency at which the UIL processor is scheduled -- and thus the frequency at which it gets a chance to act on a UIL sequence. Depending on the purpose of a UIL environment, this rate may vary. The rate may be very rapid in the case of an environment that exercises fine-grain control over fast-moving operations, but other environments can afford to work in a more leisurely fashion. The iteration rate is the frequency at which the condition specified in a VERIFY (or WHEN, WHENEVER) statement is checked, and it determines the time-granularity of constructs such as WAIT (or EVERY).

RECOMMENDATION: Provide mechanisms to allow the UIL writer to specify a priority and an iteration rate for each UIL environment package.

As a reviewer, I found it frustrating to have to dig into the fine print at diverse points in order to figure out what was intended by the authors of the UIL Spec in terms of the serial/parallel arrangement of UIL entities. This issue is of fundamental importance. It would make it easier on readers if it were dealt with explicitly in the UIL Spec. Doing so might have the additional benefit of illuminating the contradictions that may unwittingly have been built into the UIL architecture.

RECOMMENDATION: Modify the UIL Specification so that it explicitly describes the concept of the language in terms of the serial/parallel operation of the various entities that are proposed.

## 1.2 Comments on "Procedure" (Section 3.7, 4.17)

The main problem I have with the UIL concept of "procedure" -- apart from its relationship to the "environment", as discussed above -- is its name.

The term "procedure" is used in computer programming languages to denote subroutines that are called by a higher-level entity (program) that establishes the stream of execution of which the procedure call forms a part. The analogy is not appropriate for UIL because UIL "procedures", although they exist within a context established by an "environment", are not called by the environment. UIL "procedures" may establish their own independent, parallel streams of execution rather than participating in another stream as "software procedures" do.

Similarly, within the Space Station Project the term "crew procedure" is used to describe actions taken by the on-board crew. UIL procedures interact with crew procedures, but they are not identical. At the recent PDR in Huntington Beach at least one RID (3586) called for the distinction between these types of procedure to be made clear, and other RIDs showed evidence of the misunderstandings that are created if it is not.

For these reasons it would be helpful to find a term other than "procedure" for referring to UIL entities. I suggest the word "sequence". The term "sequence" implies the serial (or "sequential") nature of UIL entities, each of which may in turn coexist in parallel with other serial sequences.

The use of the term "sequence" is also appropriate because it is the nature of UIL to provide an outer shell for the SSF DMS that provides the functions of a "master sequencer". That is, the entities written using UIL collectively provide a sequencer than will supplement and replace the role of the crew and ground controllers in sequencing the every-day operation of the space station's systems. Therefore it is appropriate that each of the entities that make up the UIL be called a "sequence".

In the remainder of this review I often use the word "sequence" instead of the word "procedure" because of its superiority in describing the function of a UIL entity.

RECOMMENDATION: Use the word "sequence" instead of the word "procedure" to denote each of the serial streams, within an "environment", that constitute the basic unit of UIL processing.

For the sake of visibility a UIL sequence designed to be "called" for in-line execution should be distinguished from one designed for concurrent execution and invoked by the PERFORM statement. Therefore I suggest that the concept of the "subsequence" be introduced. A "subsequence" would be like a "sequence" except that it is designed for in-line execution and thus would never create its own stream of execution.

RECOMMENDATION: Create an entity called the "subsequence" that would be incorporated within the stream of execution of the sequence that "calls" it using the CALL statement.

## 1.3 Comments on "Events" and "Event Handlers" (Sections 3.8, 4.18, 4.19)

Event handlers are described as constructs "used to indicate a condition of which an environment is to be made aware", without the environment "having to anticipate or wait for the event to occur". It is also stated that "the appropriate event handler is invoked automatically when the environment is notified that an event has occurred".

From these two statements it appears that an "event handler" is precisely the form in which an environment does "anticipate or wait for" the occurrence of a specific condition.

Within the context of the UIL Spec it is not entirely clear what is meant by an "event". An "event" is described as a class of object that "can trigger event handlers" -- a circular definition. The only place where I found a reference to the origin of events was in the section on the SIGNAL statement

(4.19.4.1). From this section it appears that an "event" is something "signalled" by a UIL sequence -- that is, a device for communication among sequences.

Evidently what we think of as "mission events", such as those that emanate from GN&C, are not the same thing as UIL events. Such events would apparently have to be detected and signalled by a UIL sequence in order to trigger any "event handlers" that may be waiting for them. This seems to introduce an extra step. Wouldn't it be better for the sequence that needs the information to have constructs allowing it to see the "mission event" directly?

From the examples of events found in the document it appears that an event is considered to be something like a boolean. Nothing serves to indicate that, for example, the drop in a battery voltage from 26 to 25 volts is in itself an event -- yet one should have the ability to write a UIL sequence that can respond to such an occurrence at any time.

In other words, the event handler establishes a method of dealing with one special class of object that is different from the way all other kinds of object are dealt with. UIL will require access to a whole menagerie of object/attributes -- valves, temperatures, switches, software, etc., etc. Access to each type of object may require special mechanisms at the implementation level, but these mechanisms are made transparent to the UIL writer so that he can deal with all objects in a uniform way. Why should a particular class of object called "the event" be an exception?

It appears that both the "event handler" and the concept of a UIL "event" itself are special cases, and as such perhaps not too useful. A UIL sequence will need to detect and act upon conditions that do not seem to fall within the UIL Spec's conception of what an event is, including cases where AND's and OR's are used to form compound conditions in terms of objects that may or may not be "events".

Another shortcoming of the "event handler" is that it is designed as a stand-alone structure. It apparently cannot be nested within a sequence or within another event handler. Also, the event handler contains no features for turning itself on or off, and so an external procedure must do so. The ENABLE and DISABLE commands are provided for this purpose. Since "event recognition shall only be enabled/disabled for the environment in which the command is issued" (page 4-50), the procedure that issues the ENABLE or DISABLE command must be within the same environment as the event handler.

If we consider the need for a separate procedure to disable and enable an event handler, in relation to the implication that each environment contains only one active procedure (see above), we are led to the conclusion that each event handler and its auxilliary will constitute a separate environment. Is this really what is intended?

The weakness of the current concept of "event handlers" is demonstrated by the fact that (as illustrated in section 2.7 below) UIL writers may tend to use such constructs as the VERIFY statement inside a REPEAT loop to detect events, because such a construct can recognize compound and non-event conditions (as well as simple events), can turn itself off, and can be incorporated within a larger sequence.

If the concept of "event handler" is primarily motivated by the need to establish concurrency, this can be solved more generally by allowing concurrency among sequences (procedures) within an environment as proposed above.

I could go on, but I think I have said enough to demonstrate that the whole notion of event handlers needs more thought, or at the very least a clearer exposition in the UIL document.

I would prefer to define a new construct that "generalizes" the event handler. Since more than "events" are detected by such a construct, the name "event handler" is unsuitable. That is just as well because the nomenclature "event handler" smacks of computer jargon, and may be unnecessarily alienating to users who lack a computer science background.

I suggest that the special-case concept of the "event handler" be replaced by a general-purpose structure based on the English word "whenever". Such a WHENEVER statement is included in the new constructs described in Part 2 of this review.

The proposed WHENEVER construct should make the following improvements on the existing concept of the "event handler":

(1) Like the event handler the WHENEVER construct will cause a block of statements to be executed whenever a specified condition comes into existence. Unlike the event handler, the condition may be stated in terms of objects and attributes other than a single event. That is, the condition sought by a WHENEVER statement can be described by a logical expression written in terms of any available objects.

(2) The WHENEVER construct should be able to stand alone in its own UIL sequence, but it should also allow itself to be incorporated into a larger sequence at any point, or to be nested within other constructs. This allows processing to occur before or after the loop that waits for and acts on a condition.

(3) The WHENEVER construct should include a modifier to permit it to be terminated upon the occurrence of a condition (the BEFORE clause), or upon the expiration of a time interval (the WITHIN clause). There should also be the capability (the WHEN construct) to provide one-shot event handling capability.

The WHEN and WHENEVER constructs are described more completely in Part 2. The following example shows how they would be used to perform the "event handler" function. Consider the case on page 3-71:

```
Handler for DAY TERMINATOR CROSSING is
   Turn On SHUNT CURRENT REGULATOR
```

```
    Verify SHUNT CURRENT is within LIMITS Within 30 SECONDS
       Then
          Perform SOLAR PANEL RECONFIGURATION
       Otherwise
          Remove BATTERY1, BATTERY2 From PRIMARY POWER BUS
          Issue BATTERY OVERCHARGE WARNING MESSAGE to MPAC1
    End Verify
 End DAY TERMINATOR CROSSING
```

Using the new constructs proposed in this memo this case would be written as follows:

```
 Whenever DAY TERMINATOR CROSSING
    Turn On SHUNT CURRENT REGULATOR
    When SHUNT CURRENT is within LIMITS
       Within 30 SECONDS
          Perform SOLAR PANEL RECONFIGURATION
       Otherwise
          Remove BATTERY1, BATTERY2 From PRIMARY POWER BUS
          Issue BATTERY OVERCHARGE WARNING MESSAGE to MPAC1
    End When
 End Whenever
```

The construct could be written with the WHEN statement instead of the WHENEVER statement if it is desired that the block of actions be performed only the next time that a day terminator crossing occurs rather than when<u>ever</u> it occurs.

In addition, if written using concepts such as WHEN or WHENEVER the "event handler" function may be incorporated within the context of a larger UIL sequence, and may even be nested inside other constructs. For example the following case:

```
 When TIME = 1995/12
    Whenever DAY TERMINATOR CROSSING
       Until TIME = 1996/1
          ...
    End Whenever
 End When
```

establishes a sequence that would act on a day terminator crossing only during December of 1995. Providing this function using the existing constructs would require an external procedure to enable and disable the operation of the event handler on the appropriate dates.

RECOMMENDATION: Remove the concept of the "event handler" from the UIL language because its functions can be performed in a more general way by constructs based on the English words "when" and "whenever" that can (1) act on other objects besides events, (2) be incorporated directly into UIL sequences, and (3) include modifying clauses to turn themselves off. These constructs are described in Part 2 of this report. Also, eliminate the ENABLE and DISABLE commands that are provided for the special case of the event handler.

In section 1.5.3.2.2 the UIL Spec admits that "many of the issues regarding binding of proceaures and event handlers remain to be worked out". Some of the problems I have noted in this and the preceding sections are probably traceable to this statement. I believe that these issues can be solved by eliminating the event handler as a special case, by allowing multiple concurrent procedures within an environment, and by using the environment for the "binding" of procedures as well as for the "scoping" of procedures to an appropriate subset of objects.

## 1.4 Comments on "Step" and the GOTO Statement (Section 3.5.5.2)

The lowest level of the hierarchy of UIL entities is the "step". Multiple steps make up a "procedure".

The main function of the concept of "step" seems to be to allow points in a UIL procedure to be named such that control may be transferred to them by use of the GOTO statement.

Use of GOTO is regarded as dangerous within the context of programming languages because it diminishes visibility by allowing a jump to occur for which the return, if any, is not specified in a visible way. The use of GOTO statements may create blocks of code that have one entrance, but multiple exits. This will tend to create what is sometimes called "spaghetti code".

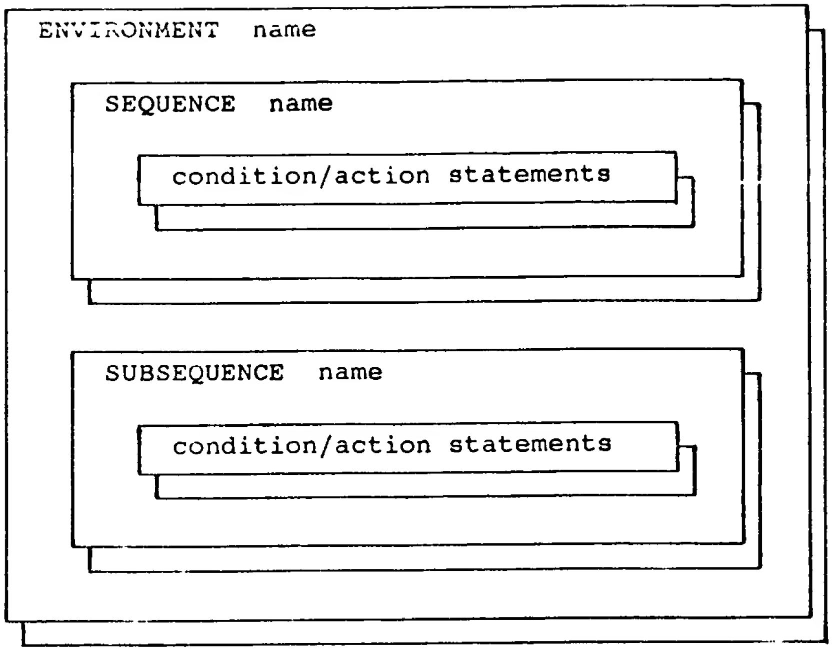
Programs that use GOTO are more difficult to test and verify that ones that do not. "Structured" programming languages forbid the use of the GOTO statement and establish alternative constructs to perform its legitimate functions. The EXIT statement, for breaking out of loops, and the CALL statement, which creates a jump but implies that the return will be to the next statement, are considerably less problematic than GOTO.

These arguments apply equally to the UIL language, which is also used to create "programs" (or "scripts") that will have to be verified before use. If the existing UIL syntax seems to require the existence of the GOTO statement, this should be taken as a sign that the syntax itself is inadequate.

RECOMMENDATION: Remove the unnecessary concept of the "step" from the UIL language. The next level of the hierarchy below the "sequence" and "subsequence" shall be the individual action and condition statements, and the groupings of statements formed by the various constructs. Remove the GOTO statement from the UIL language. Its legitimate purposes can be performed by the CALL statement or the EXIT statement.

## 1.5 Summary of Recommended Changes in the UIL Hierarchy

If the recommendations proposed in sections 1.1 - 1.4 are adopted, the hierarchical organization of the UIL will be as described in the following picture:



The concept "environment" is basically unchanged from the document under review. Its main purpose is to establish the "scope" of its constituent "sequences". Multiple environments having varying scopes and purposes would coexist with each other. Different environments may be executed in different processors within the space station DMS, and may be executed at varying priorities and iteration rates as determined by their purposes.

The concept "sequencer" is used to denote a collection of condition and action statements that would execute serially (or in loops explicitly set up by its constituent statement constructs). The multiple "sequences" within an "environment" would execute in parallel with each other.

The concept of "subsequence" is introduced. A "subsequence" would be called by a "sequence" and would be incorporated into the serial stream of execution established by the calling sequence, to which it would return when finished.

The "event handler" is eliminated. Its function is provided by more general constructs such as WHEN and WHENEVER that can be incorporated anywhere within sequences and subsequences and recognize other conditions besides events.

Any document describing the UIL should explicitly introduce the serial/parallel nature of the UIL at an early point because that concept is an important prerequisite for understanding the details of the UIL.

The nomenclature of "environments", "sequences", and "subsequences" as defined here will be used below in this review.

## 1.6 Comments on "Command Qualifiers" (Section 3.5.4.2)

The essence of the User Interface Language is to cause actions to occur, as required, over the course of time. This implies that UIL will contain two basic types of statement:

(1) "Condition statements" that will state preconditions that must exist before actions will be carried out, and

(2) "Action statements" that specify the actions or commands themselves.

Both condition and action statements deal with objects and their attributes. The first specifies conditions in terms of objects. The second specifies actions to be taken with respect to objects. However the roles are fundamentally different. Conceptual clarity will be enhanced if the two types of statement are separated.

RECOMMENDATION: Establish a clear distinction between "condition" statements, which establish the conditions that must occur before commands are performed, and "command" (or "action") statements, which specify the actions to be taken.

In fact, the existing UIL already conforms to this recommendation in most cases. The most glaring exception is the case of the "command qualifiers" that form part of a "command" or "action" statement. Some of these qualifiers provide the desirable capability of modifying the action itself. It is those command qualifiers that may be used to affect "the time at which an action is to occur", that are of concern.

These are: AFTER, AT, BEFORE, EVERY, UNTIL, FROM, TO, and WITHIN -- <u>when</u> they are used to determine the time at which or during which the action occurs.

There is a certain ambiguity between the use of these words to describe the time at which an action is to occur, and their use to describe characteristics of the action itself. It seems fairly clear that the statement:

```
Set TEMPERATURE of LAB MODULE to 72 DEGF At 301/1991-9:00
```

uses the AT clause to specify the date and time at which an action (setting the lab temperature) should occur. However the statement:

```
Plot RADIATOR TEMPERATURE From 301/1991-9:00 To 301/1991-10:00
```

is less clear. Is it an immediate command to plot stored data points that lie between the two times, or is it intended to provide a pair of commands to start and stop the plotting of data at the two times?

Consider the case:

```
Display ECLSS STATUS At 301/1991-4:30
```

If the specified time is in the past, this statement would appear to be a command to immediately display the ECLSS status pertaining to the time. If the specified time is in the future, one would tend to interpret the statement as a command to display the status when the time arrives. Such ambiguities do not seem healthy.

Note that a semantic distinction is being made here between "qualifiers" that state a precondition for an action to occur, and other qualifiers that state qualifications applicable to the action itself. In my view only the latter should properly be called "qualifiers". The statement of a precondition for an action is not really a qualification of the action.

Further imprecision is inherent in the command qualifiers BEFORE and AFTER. These are described in the text as meaning "a date and time before/after which an action is to occur". Is it really the intention to give the UIL processor such a wide latitude to decide when to perform an action?

Such ambiguities are avoided if the use of qualifiers within a "command" is confined to cases in which they are being used to modify the action itself, and if separate condition statements are used to determine the timing of the action.

For example, let us assume that the following case (taken from page 3-59) means that a running plot of radiator temperature is to be terminated at a certain time:

```
      Stop Plotting RADIATOR TEMPERATURE After 1991/301-12:30
```

I believe it would be clearer and more rigorous to write:

```
      When TIME = 1991/301-12:30
         Stop Plotting RADIATOR TEMPERATURE
      End When
```

Such a case as the following:

```
      When TIME = 301/1991-10:01
         Plot RADIATOR TEMPERATURE From 301/1991-9:00 To 301/1991-10:00
      End When
```

would then clearly mean that at 10:01 the radiator temperature data from the previous hour should be plotted.

Another consideration that influences this choice is that it will usually be the case that more than one action will occur upon the realization of a given condition. Accordingly:

```
      WHEN Time = 1991/301-12:30
         Stop Plotting RADIATOR TEMPERATURE
         Print Plot of RADIATOR TEMPERATURE
      End When
```

Seems easier and more elegant than:

```
    Stop Plotting RADIATOR TEMPERATURE After 1991/301-12:30
    Print Plot of RADIATOR TEMPERATURE After 1991/301-12:30
```

It is not clear whether the time could be omitted from the second statement, on the supposition that the second statement will not be encountered until the first is executed.

Another reason for finding ways other than the command qualifier for making actions occur when they are wanted is that the existing command qualifiers designed for stating the preconditions for an action (AFTER, BEFORE, AT, etc.) deal only with time. It will also be necessary to specify actions that are to occur under more complicated conditions that cannot be specified in terms of time alone. A statement construct such as the WHEN statement (described in Part 2) permits the use of any logical expression, not just one relating to time, to control when an action occurs.

For example, it should be possible to write:

```
    When RADIATOR TEMPERATURE > 150 DEGF
       Start Plotting RADIATOR TEMPERATURE
    End When
```

Another disadvantage of the use of the "command qualifier" to specify the times at which actions should be taken is that it subjugates the condition information to the action information, and specifies it <u>after</u> the action in question. In fact, the satisfaction of the condition must come <u>before</u> the execution of the action and the syntax should reflect this fact unless there is a good reason otherwise. Since multiple actions may follow from the satisfaction of one condition (as illustrated above) it is all the more appropriate for the syntax to reflect the precedence of the condition over the action.

Other "command qualifiers" not related to the time or condition under which an action should occur -- in other words those that are used to modify the action itself -- will still be useful and should be retained.

RECOMMENDATION: Remove from the list of command qualifiers those that have to do with the time at which the command will be performed. Confine command qualifiers to the function of specifying qualifications pertaining to the action itself. The lost functionality will be replaced in a more general way by statement constructs such as the WHEN statement defined in Part 2.

## <u>1.7 Comments on the VERIFY Statement (section 3.5.6.1)</u>

The simple VERIFY statement, as currently defined, does not wait for the condition to occur. It is simply another way to write an IF statement for immediate evaluation. The following example from page 3-57:

```
Verify PUMP1 is ON
   Otherwise
      Issue PUMP FAILURE ALERT MESSAGE
End Verify
```

could be written as:

```
If PUMP1 is OFF
   Issue PUMP FAILURE ALERT MESSAGE
End If
```

or, if there is no IF statement, it could be written as follows using the CHOICE construct defined in section 3.5.6.2 of the UIL Spec:

```
Choice PUMP1
   When OFF Then Issue PUMP FAILURE MESSAGE
End Choice
```

Since it provides redundant capability, the simple VERIFY statement could be eliminated.

(Note that in my opinion it is counterproductive for UIL to eschew the simple IF statement. The meaning of the IF statement is obvious and familiar, and thus it serves the UIL goals of ease of use and lack of ambiguity. An argument for inclusion of the IF statement is made in section 2.8 below.)

Rather than eliminate the simple VERIFY statement it might be better to change the concept such that VERIFY continues to test the condition indefinitely, dropping through only when the condition is fulfilled. This would increase its usefulness -- while incidentally making it identical to the WHEN construct described in Part 2. Without this change, the kluge-clause "Within INFINITY" would be necessary to make VERIFY wait indefinitely.

RECOMMENDATION: Eliminate the simple VERIFY statement as a way to perform an immediate test. This function is redundant because it can be performed by other constructs such as CHOICE or IF.

When amplified by a WITHIN clause, the VERIFY statement provides a construct with considerable usefulness. My only problem with the VERIFY/WITHIN construct concerns the connotations of the word "verify".

The word implies that the VERIFY construct is geared to the "verification" of a previous action, or an anticipated condition, rather than to the triggering of the next action. The quoted examples reinforce the impression that VERIFY is intended for the confirmation of previous actions. The conditions for the next action may include, but not be limited to, the verification of a previous action.

In other words, an action may be taken and a VERIFY/WITHIN statement then executed that will issue an error message if the action has not taken effect in a certain period of time. The <u>next</u> action may have other preconditions in addition to the successful verification of the preceding action. If these additional preconditions were added to the logical expression in the verify statement, the error message might be issued even though the previous action was properly carried out, because the additional precondition for the next action is not yet present.

To perform this function adequately, it would be necessary to use two VERIFY statements, the first to verify the previous action and the second to wait for the next action's precondition. The use of the word "verify" in the second case would be a misnomer because the precondition sought is not a verification of some previous action.

For these reasons, I favor the use of a more neutral term such as "when" instead of the term "verify", which carries potentially misleading connotations when not being used to "verify" a previous action or expected condition.

A WHEN construct is defined in Part 2 that is a "superset" of the VERIFY construct. It can do everything VERIFY/WITHIN can do. In addition, it can be terminated upon occurrence of any condition that can be specified as a logical expression (not just a time interval), and it can be used without a terminating clause to establish an indefinite wait for a condition to occur.

I do not think that suggestions of this kind should be dismissed as based on "personal preference". The elimination of constructs that carry the seeds of misunderstanding will make UIL easier to use over its long lifetime, and may help prevent costly mistakes. More general constructs may actually be easier to implement, and thus provide extra capability free of charge.

Using the WHEN statement, the example given on page 3-57:

```
Verify PUMP1 is ON Within 10 Seconds
   Then
      Set SPEED of PUMP1 to 1234 RPM
   Otherwise
      Issue PUMP FAILURE ALERT MESSAGE
End Verify
```

would be written as follows:

```
When PUMP1 is ON
   Within 10 Seconds
      Set SPEED of PUMP1 to 1234 RPM
   Otherwise
      Issue PUMP FAILURE ALERT MESSAGE
End When
```

The word "when" works just as well as "verify" when the intent is to verify a previous action, and works better when the intent is to establish preconditions for the next action that are independent of the previous one.

RECOMMENDATION: Discard the VERIFY/WITHIN statement and include the more general WHEN construct defined in Part 2 instead.

## 1.8 Comments on the CHOICE Statement (section 3.5.6.2)

The CHOICE construct also embodies a decision that could be written by means of the IF statement. The following example from page 3-59:

```
Choice PRIMARY PUMP
   When ON      Then Set Speed of PRIMARY PUMP to 1234 RPM
   When OFF     Then Turn On  BACKUP PUMP
   When STANDBY Then Turn Off BACKUP PUMP
End Choice
```

could be written as follows:

```
If PRIMARY PUMP is ON
   Set Speed of PRIMARY PUMP to 1234 RPM
Else If PRIMARY PUMP is OFF
   Turn On  BACKUP PUMP
Else If PRIMARY PUMP is STANDBY
   Turn Off BACKUP PUMP
End If
```

As compared to IF the CHOICE statement offers greater conciseness when dealing with a choice of a large number of conditions, where a long string of ELSEIF's might be required. Therefore CHOICE probably should be retained even if the IF construct is added.

A more serious problem affecting the CHOICE construct has to do with the use of the word "when" in the subsidiary clauses that determine what action is taken as a result of the various possible states of the so-called "determinant". This usage is consistent with the fact that in the English language the word "when" (and sometimes "where") is often used loosely as a synonym for "if".

However, since the UIL language functions within the domain of time, the use of "when" may be misleading, especially if an explicit WHEN construct is added to the language, as recommended.

If "when" is used to specify a check that is performed immediately within a CHOICE construct, the reader of a UIL sequence may erroneously think that the actions specified in the "when" clauses are to be performed at a future time when (or whenever) the conditions are fulfilled. Since the tests implied by the CHOICE construct are made immediately (waiting only for the availability of valid data), the word "if" would be more appropriate.

The word "when" is a valuable one for use in situations where temporality is involved, and it should be reserved for such cases rather than being squandered in a situation where another word would work just as well.

It is evident that the use of the word "when" is influenced by the "case" statement of the Ada language, where "when" is used in the sense of "if". In Ada no ambiguity is created, since Ada unlike UIL is a programming language not intended for the control of actions taking place during the course of time. We should strive to make UIL similar to "natural, spoken English", as opposed to a programming language. We should not assume that astronauts, or even ground controllers, are familiar with programming languages such as Ada. For such persons the use of "when" may be especially misleading.

RECOMMENDATION: Find an alternative, such as "if", for the use of the word "when" in the clauses that determine the action to be taken for each state of the "determinant" in a CHOICE construct.

Another problem with the CHOICE statement is that a series of actions may be required in consequence of a particular state of the determinant. No provision is made for nesting multiple statements within each of the choices. This problem goes away if the IF/ELSE construct is used instead of the CHOICE construct because IF/ELSE permits nesting. If CHOICE is retained, some way should be found to provide the nesting function where needed.

RECOMMENDATION: Find some way to nest multiple action and condition statements within each of the alternatives lying within the CHOICE statement.

Another confusing aspect of the CHOICE construct has to do with the statement that "if a CHOICE statement has no WHEN clause it must have an OTHERWISE clause". Since the naming of a "determinant" on the CHOICE line does not in itself establish a logical expression that can be evaluated, the meaning of an OTHERWISE clause without a WHEN clause is unclear. No example is given of such a case.

RECOMMENDATION: Do not allow an OTHERWISE clause unless a WHEN clause is present. Require that every CHOICE construct have one or more WHEN clauses.

### 1.9 Comments on the REPEAT Statement (section 3.5.7.1)

The REPEAT statement is used both as a statement and as a modifying clause, to cause UIL statements to be repeated.

The REPEAT statement has one very serious weakness. It does not allow the time interval at which the statements are to be repeated to be specified by the writer of the UIL sequence.

This lack would not be significant in the context of a programming language. The UIL, however, is intended for the control of actions taking place over the course of time. It may be necessary to repeat actions on time scales ranging from milliseconds to years. In the context of UIL the inability to specify the iteration rate renders the simple REPEAT statement almost useless.

Furthermore, the usage of the REPEAT statement is confusing because in some cases the REPEAT precedes and in other cases follows the statements that are to be repeated. In other cases the REPEAT statement is superfluous because the statement that it modifies already implies that a loop is performed.

In the following case, taken from page 3-61, the REPEAT statement <u>precedes</u> the block of statements that is to be repeated:

```
Repeat
   Perform BATTERY STRESS TEST
   Verify BATTERY VOLTAGE > 25.3
      Otherwise
         Goto SHUTDOWN
   End Verify
End Repeat
Step SHUTDOWN
     ...
End SHUTDOWN
```

As written, the above case would cause the battery stress test to be repeated at the UIL iteration frequency, which is likely to be considerably higher than the frequency at which it is desirable to stress test batteries. It is not sufficient to limit the REPEAT rate to some particular lower rate, because the desired frequency will be different for different cases, over a wide range.

These problems are avoided if the above case were written as follows:

```
Every 1 DAY
   Perform BATTERY STRESS TEST
   If BATTERY VOLTAGE <= 25.3
      Call SHUTDOWN
   End If
End Every
```

where SHUTDOWN is a separate UIL script that can be called as a subsequence.

In this case the time interval at which the battery is to be stress tested is explicit, and the syntax is more concise, more understandable, easier to learn, and more like English.

Note that in the example the IF statement is used in place of VERIFY in accordance with my recommendation above that the VERIFY statement be eliminated in favor of other constructs. An argument for the inclusion of the IF statement is made in section 2.8 of this memo. The CALL statement is used instead of GOTO in accordance with my recommendation in section 1.4 above.

In the following case, taken from page 3-62, the REPEAT statement <u>follows</u> a statement that is to be repeated:

```
While BATTERY VOLTAGE > 25.3 VOLTS
   Repeat
```

```
      Perform BATTERY STRESS TEST
   End Repeat
```

The text states that "the controlling logical expression shall be evaluated at the beginning of each iteration of the repeat loop". In other words, the condition embodied in the WHILE construct is evaluated as part of the loop established by a REPEAT statement that follows it.

In the following example the preceding case is rewritten by means of the EVERY construct in conjunction with the BEFORE statement:

```
Every 1 SECOND
   Before BATTERY VOLTAGE <= 25.3 VOLTS
      Perform BATTERY STRESS TEST
End Every
```

Finally, in the following case from page 3-63 the REPEAT statement is superfluous because a loop is already implied by the FOR construct that the REPEAT is used to modify:

```
For PUMP := PUMP1, PUMP2, PUMP3
   Repeat
      Verify PUMP is ON
         Otherwise
            Issue PUMP ALERT MESSAGE
      End Verify
   End Repeat
```

Both the inability of the REPEAT statement to specify the repetition interval and the confusion spawned by its varying placement and occasional superfluity argue for the elimination of the REPEAT construct.

RECOMMENDATION: Remove the REPEAT statement from the UIL. Its functions can be provided by the EVERY statement as illustrated above, and defined below in Part 2. The EVERY statement always precedes the block of statements that are to be repeated, and, in addition, allows the user to specify the interval at which they are to be repeated.

## 1.10 Comments on the WHILE Statement (section 3.5.7.2)

At first glance the WHILE construct provides a shorthand way of causing a function to be performed during the occurrence of some particular condition. However, on closer examination, there are problems.

Consider the example previously quoted in the section on the REPEAT statement:

```
While BATTERY VOLTAGE > 25.3 VOLTS
   Repeat
      Perform BATTERY STRESS TEST
```

```
End Repeat
```

First, as mentioned above in the section on REPEAT, the REPEAT statement is used to control the execution of a statement (the evaluation of the WHILE condition) that comes before it, and does not allow the frequency of repetition to be specified.

Second, according to the fine print, the "logical expression shall be evaluated at the beginning of each iteration" and if the "expression evaluates to FALSE then execution shall resume with the statement immediately following". In other words, the WHILE statement does not initially wait for the stated condition to become true. Some other construct (such as WHEN or WHENEVER) must be used to establish that the condition does obtain or else the WHILE construct will simply drop through.

Third, according to the description of the WHILE construct, processing will fall through to the statement following the end of the REPEAT statement once the WHILE condition is no longer fulfilled. This is by no means obvious by reading the example itself, which could be construed as meaning that the REPEAT loop will resume when the battery voltage again rises above the criterion.

In fact, the user of the UIL may desire either to drop through and continue, or to recycle the WHILE structure. Using the WHEN and WHENEVER constructs to specify the initiating condition, the example would be written as follows if it is to be performed only one time:

```
When BATTERY VOLTAGE > 25.3 VOLTS
   Every 1 MINUTE
      Until BATTERY VOLTAGE <= 25.3 VOLTS
         Perform BATTERY STRESS TEST
   End Every
End When
```

That is, when battery voltage exceeds the criterion, an EVERY loop is started that will operate at an explicitly defined interval until the voltage falls below the criterion.

If the actions are to be performed more than once, at any time in the future that the condition is fulfilled, it could be written as follows:

```
Whenever BATTERY VOLTAGE > 25.3 VOLTS
   Every 1 MINUTE
      Until BATTERY VOLTAGE <= 25.3 VOLTS
         Perform BATTERY STRESS TEST
   End Every
End Whenever
```

That is, whenever battery voltage exceeds the criterion, an EVERY loop is started that will operate at an explicitly defined interval until the voltage falls below the criterion. Then the construct will recycle so that it will operate again the next time that the voltage again exceeds the criterion.

RECOMMENDATION: Remove the WHILE construct from the UIL. Its functions can be performed more explicitly and flexibly using other constructs such as the ones defined in Part 2.

## 1.11 Comments on the FOR Statement (section 3.5.7.3)

I think the FOR statement is just fine as it is, except for one minor point that has already been touched on. That is, the REPEAT statement that is used in conjunction with the FOR statement is superfluous because the FOR statement already implies that a loop is created. In consequence of the removal of the REPEAT statement, the loop created by the FOR statement should be bounded by an END statement.

Therefore the example on page 3-63:

```
For PUMP := PUMP1, PUMP2, PUMP3
   Repeat
      Verify PUMP is ON
         Otherwise
            Issue PUMP ALERT MESSAGE
      End Verify
   End Repeat
```

would be rewritten as follows:

```
For PUMP := PUMP1, PUMP2, PUMP3
   If PUMP is OFF
      Issue PUMP ALERT MESSAGE
   End If
End For
```

RECOMMENDATION: Remove the REPEAT statement from the FOR construct and require that the loop created by the FOR statement be concluded by a statement reading "End For" rather than "End Repeat". The number of iterations created by the FOR statement is determined by the number of items in the index list.

## 1.12 Comments on the EXIT Statement (section 3.5.7.4)

Unlike the GOTO statement that was criticized above, the EXIT statement provides the "bounded" capability of breaking out of a loop to the statement following the end of the loop.

I am in favor of keeping the EXIT statement. However, as pictured in the following example:

```
Repeat
```

```
      Perform BATTERY STRESS TEST
      Exit If BATTERY VOLTAGE < 25.3 VOLTS
   End Repeat
```

the syntax of the EXIT/IF statement continues the potentially misleading practice of placing the action (to exit) before the condition that must be passed before the action can occur.

A second point is that it would be a good idea to add to the EXIT statement a second word that would specify what loop the EXIT is intended to break out of. Loops may be created by the WHENEVER, EVERY, or FOR constructs, and these constructs may be nested. When loops are nested, this feature would allow the EXIT statement to be used to break out of either loop. Even when loops are not nested, giving the type of the loop would promote visibility.

(Note in passing that the UIL Spec's argument against use of the IF statement, which is discussed in section 2.8 below, is in this example violated by the UIL Spec itself.)

In accordance with these points (and substituting EVERY for REPEAT), the preceding example would be rewritten as follows:

```
   Every  1 DAY
      Perform BATTERY STRESS TEST
      If BATTERY VOLTAGE < 25.3 VOLTS
         Exit Every
      End If
   End Every
```

RECOMMENDATION: Remove the IF clause from the EXIT statement and add a word that defines the type of loop that would be exited from. When executed, the EXIT statement will cause execution to break out of the innermost loop of the type specified and resume at the statement following the END statement that bounds that loop.

## 1.13 Comments on the UIL Specification Document Itself

The following comments bear upon the document used to specify the UIL language rather than upon the language itself:

* As it stands the UIL Spec moves from the detailed to the general.

  The document begins by talking about objects and attributes, and defines a plethora of elements such as text strings, bit strings, operators of various sorts, etc., etc. The document gradually works its way up through the sorts of statements that use these elements and finally to the level of environments, procedures (sequences), event handlers, and so forth, which combine the elements previously defined.

This organization submerges the broad outlines of the language in a sea of details, and makes it harder to understand how the language is really designed to work. The trees obscure the forest. It may seem to make sense to describe elements before discussing the contexts within which they are used, but in fact no reader will really be able to keep in mind all the details without first being motivated by an understanding of the context.

It would be much better, in my opinion, to turn the document upside-down: to start with the establishment of the upper-level serial/parallel organization, using place holders for lower-level details, and discuss these details later. It is more important for the readers of the UIL Specification to grasp the architecture than the details.

The importance of readability is not confined to user's guides. Readability is also important in documents used to describe a preliminary design, because the effective review of such documents by persons other than the authors and the eventual builders is an important step in the evolution of a good design.

The dearth of readable documentation is certainly not confined to the user interface area. It is a serious program-wide problem. If the documents are not clear how do we know that the thinking behind them is?

* The document uses bold-face type when it uses reserved words within a text description. This is unsatisfactory because the distinction between regular and bold-face type is completely lost as the document is xeroxed through several generations, as will inevitably occur with such a document. This was true of the copy I reviewed, and occasionally I was temporarily misled as a result. I suggest that reserved words be written in the text as either all-caps or in italics. Within this review I have used capitalization when referring to reserved words in the text.

## 2.0 PROPOSED NEW CONSTRUCTS

This part of this document describes more formally the new constructs that have appeared in some of the examples in Part 1. I believe that these constructs provide a more straightforward system than the existing constructs for specifying the preconditions under which certain actions should be taken.

### 2.1 Note on TIMELINER

My comments on the user interface language are based in part on my experience in building and using a language called TIMELINER that is used by various groups at CSDL to provide inputs to batch simulations, including the role of providing a "paper pilot". TIMELINER was conceived from the beginning as a time-domain tool.

TIMELINER is not as comprehensive as UIL in the area of actions, but I believe it is considerably more straightforward in the area of putting actions together to achieve particular functions over time.

Most of the UIL constructs proposed in this memo are based on the TIMELINER language. Coupled with the comprehensive object/attribute structure and the capable action statements described in the document under review, these constructs will improve the capability of the UIL.

TIMELINER has accumulated a great deal of experience over the nine years of heavy usage that it has received at CSDL, and has evolved in response to user experience. One example of TIMELINER's evolution is the IF/ELSE construct, which was added to TIMELINER at the request of its users.

At CSDL TIMELINER "scripts" consisting of as many as 120 sequences (including callable subsequences), with a total length of more than 2000 statements, have been routinely used for the control of simulations. The TIMELINER language is very popular with its users, of whom Peter Kachmar is probably the most experienced.

In terms of implementation, TIMELINER functions as an assembler/interpreter language. At initialization time syntax is checked and the existence of the objects that are named is verified. Error messages are issued where necessary. The information in the statements is stored in a tabulated, numerical form that facilitates run-time processing. During the run, using the stored information, the script is interpreted by a procedure attached to the high-frequency executive. Even when multiple parallel sequences are being processed, the CPU burden is minimized because on the typical pass only two questions need to be asked about each sequence: is it active, and is the condition currently being sought satisfied.

I believe it would be a revealing exercise -- suitable for a student -- to attempt to translate a TIMELINER script taken from an actual application into the UIL language as currently specified. This would show which language offers a cleaner way to organize such functions.

RECOMMENDATION: The "background" section (1.1) of the UIL Spec enumerates the various languages that have influenced the design of the User Interface Language. Those charged with designing the UIL should also become aware of the CSDL TIMELINER language, the features and philosophy of which have the potential to improve the capability of the UIL.

I would be happy to give a pitch on the subject of the TIMELINER language to any audience interested in the design of the UIL for the space station.

## 2.2 Summary of the Existing Constructs for Stating Action Preconditions

As described in the document under review, the UIL contains five methods for causing actions to occur under certain conditions: command qualifiers, the WAIT statement, the VERIFY statement, the WHILE statement, and event handlers.

(1) "Command qualifiers" are an unsatisfactory way to supply this function because they are often ambiguous, they may have to be repeated if several actions are to occur under the same conditions, they are limited to conditions that can be stated in terms of time, and they are awkwardly placed after the actions for which they state the preconditions. They violate the principle of the "separation of concerns" since action and condition functions are combined in one statement. (See section 1.6.)

(2) The existing WAIT statement applies only to conditions stated in terms of an interval of time, and there is no simple way to cancel a wait if another condition intervenes.

(3) The VERIFY statement has the advantage that unlike "command qualifiers" or the WAIT statement the condition for the next action can be stated generally in terms of any logical expression. However, the use of VERIFY to control actions over time is geared to the special case of "verifying" the performance of an action previously requested. Successful verification of a previous action is not always a sufficient precondition for performance of the next action. (See section 1.7.)

(4) The WHILE statement is useful for stopping the repetition of an action when a condition is no longer present, but it is not suitable for initiating the action. It does not wait for the specified condition to occur, but will drop through if it is not present on the first pass. (See section 1.10.)

(5) "Event handlers" are not a general method because they do not lend themselves to specifying compound conditions or conditions described in terms of objects/attributes other than events. In addition, they are not designed to be incorporated in a flexible way within serial sequences and they do not have the ability to turn themselves off. The nomenclature "event handler" may be off-putting to users unfamiliar with programming languages. (See section 1.3.)

These five methods comprise a medley of special cases: a "verify" statement that can be limited by a time interval but not by a logical expression, an "at" capability that is limited to the "command qualifier" clause of an action statement, an ability to perform an action "whenever" a condition occurs that is limited to a particular class of objects known as "events". They do not form a systematic whole that will easily be grasped by users of the UIL.

## 2.3 The WHEN and WHENEVER Constructs

The function of specifying the conditions under which actions are to be taken is so central to the design of a language for controlling actions over the course of time that its design deserves more thought. We should seek to define a general set of constructs that will provide all the functions of the existing "special cases", while adding additional capability at no extra charge.

To suggest such a new approach, it may be helpful to quote the sequences that I regard as the essence of the UIL task:

```
When  condition
   action
   action                                Whenever  condition
When  condition             OR              action
   action                                   action
   action
[etc.]
```

In the case on the left, execution proceeds serially through the sequence. When the first condition is fulfilled, the first set of actions is performed. When the next condition is fulfilled, the next set of actions is performed, and so on.

In the case on the right, a set of actions is performed whenever a condition is fulfilled. Then the loop "recycles" and waits for the next occurrence of the condition, whereupon the actions are repeated.

At any given time, two or more such sequences may be waiting for conditions to occur that are independent of each other and may occur in any order. The two types may occur in combination. For example, the left-hand sequence may be nested within the right-hand case.

In both cases the "condition" may be stated in a general way as a logical expression involving any objects and attributes, including but not limited to time and events. Each such sequence is executed serially, while multiple sequences coexist in parallel. Sequences can interact with each other by means of signals, and by initiating, canceling, suspending, or resuming other sequences.

I claim that these two sequences typify the functions that the UIL, in its compiled form, will be called upon to perform.

If this is the case, there is no better place to start than to define two new UIL constructs: the WHEN statement and the WHENEVER statement.

```
when_statement  ::=  when  logical_expression
                       [before_clause | within_clause]
                          block_of_statements
                       [otherwise clause]
                     end when
```

When execution of a UIL sequence reaches a WHEN statement, the logical expression will be checked upon each iteration of the UIL logic. If the stated condition has been fulfilled, the block of statements following the WHEN statement (usually consisting of action statements) is executed.

The WHEN construct must be closed by means of an END statement. Other condition constructs may be nested within the WHEN construct. When the block of statements has been completed execution proceeds to the statement following the END statement.

The WHEN statement may be thought of as a gate along a straight road that opens when a specified condition is met. The gate must open to allow execution to proceed further.

The WHEN statement may be accompanied by a BEFORE or WITHIN clause (defined separately below) that will terminate the construct if the WHEN condition does not occur before another specified condition, or within a specified time interval. If a BEFORE or WITHIN clause is present, an OTHERWISE clause may be included.

An example of the use of the simple WHEN statement is as follows:

```
When  TIME >= WAKE_UP_TIME  or  CREW ALARM
   Issue WAKE UP CALL
   Prepare COFFEE
End When
```

```
whenever_statement  ::=  whenever  logical_expression
                           [before_clause | within_clause]
                              block_of_statements
                           [otherwise_clause]
                         end whenever
```

When execution of a UIL sequence reaches a WHENEVER statement, the logical expression will be checked upon each iteration of the UIL logic. If the stated condition has been fulfilled, the block of statements following the WHENEVER statement (usually consisting of action statements) is executed.

The WHENEVER construct must be closed by means of an END statement. Other condition constructs may be nested within the WHENEVER construct. When the block of statements has been completed execution returns to the WHENEVER statement. Upon the next off-to-on transition of the stated condition the block of statements will again be executed.

The WHENEVER statement may be thought of as a gate along a circular road. The gate must open to allow execution to proceed further. Following execution of the block of statements following the WHENEVER statement, the same gate again presents itself.

The WHENEVER statement may be accompanied by a BEFORE or WITHIN clause (defined separately below) that will terminate the loop upon occurrence of another specified condition, or upon expiration of a time interval. If a BEFORE or WITHIN clause is present, an OTHERWISE clause may be included. Unless the WHENEVER construct is qualified by a BEFORE or WITHIN construct (or contains an EXIT statement) the loop will continue until the sequence of which it forms a part is externally deactivated.

An example of the use of the simple WHENEVER statement is as follows:

```
Whenever  COLLISION WARNING
   Issue CREW ALARM
End Whenever
```

---

The WHEN and WHENEVER constructs allow the UIL to perform certain functions simply that would require combinations of the existing UIL constructs. For example WHENEVER is more flexible that an "event handler". Furthermore, the WHEN statement has no exact equivalent in the present UIL; its function can only be performed by somewhat inelegant forms of the WHILE or VERIFY statement. Section 2.7 of this memo contains examples of functions that are easier to write with the new constructs than the old ones.

## 2.4 The WAIT and EVERY Constructs

As argued above in section 1.9 the existing REPEAT statement is almost useless within the time domain of the user interface language because it does not allow the user to specify the time interval at which the repetition should be performed. Rather than simply adding a time interval to the REPEAT statement it seems preferable to use the word "every", as in the statement, "every 5 minutes check the roast".

An EVERY statement is defined below. A WAIT statement is also defined here that is slightly different from the existing UIL WAIT statement.

---

wait_statement ::= **wait** *timing*_expression

```
                      [before_clause | within_clause]
                         block_of_statements
                      [otherwise_clause]
                   end wait
```

When execution of a UIL sequence reaches a WAIT statement, the sequence stops until the specified amount of time has elapsed. When the specified interval has passed execution proceeds to the next statement.

The block of statements following the WAIT statement must be closed by means of an END statement. Other condition constructs may be nested within the WAIT construct. When the block of statements has been completed execution proceeds to the next statement.

The WAIT statement may be accompanied by a BEFORE or WITHIN clause (described separately below) that may cause early termination of the waiting period. If a BEFORE or WITHIN clause is present, an OTHERWISE clause may be included.

An example of the use of the simple WAIT statement is as follows:

```
      Wait  5 SECONDS
         Issue CREW ALARM
      End Wait
```

Note that the WAIT statement already exists in the UIL (described in section 3.5.5.3). WAIT is included here for completeness and because certain modifications have been made, such as the option of using a BEFORE or WITHIN clause in conjunction with it.

---

```
every_statement  ::=  every  timing_expression
                         [before_clause | within_clause]
                            block_of_statements
                         [otherwise_clause]
                      end every
```

When execution of a UIL sequence reaches an EVERY statement, execution passes immediately to the block of statements following the EVERY statement. Following execution of those statements, execution returns to the EVERY statement and waits until the specified amount of time has elapsed since the previous encounter with the EVERY statement. Then the statements following the EVERY statement are again executed, and so on.

The block of statements following the EVERY statement must be closed by an END statement. Other condition constructs may be nested within the EVFRY construct.

The EVERY statement may be accompanied by a BEFORE or WITHIN clause (described separately below) that will terminate the loop. If a BEFORE or WITHIN clause is present, an OTHERWISE clause may be included. Unless the EVERY statement is qualified by a BEFORE or WITHIN statement the loop will continue until the sequence of which it forms a part is deactivated.

An example of the use of the simple EVERY statement is as follows:

```
Every  60 SECONDS
   If  BATTERY VOLTAGE  <=  25.3 VOLTS
      Call SHUTDOWN
   End If
End Every
```

WAIT and EVERY are analogous to WHEN and WHENEVER. Like WHENEVER, the EVERY statement creates a loop. Like WHEN, the WAIT statement waits before proceeding. Unlike WHEN and WHENEVER the WAIT and EVERY statements require that a time interval be specified rather than a logical expression.

## 2.5 The BEFORE and WITHIN Clauses

It will sometimes (perhaps always) be important to allow for the case in which the condition being awaited by a UIL sequence does not occur before the need for it is obviated by the occurrence of some other condition or the passage of time.

The present UIL recognizes this need in the case of the VERIFY/WITHIN construct, where the attempt to verify is suspended upon the expiration of a time interval. This capability should be generalized, in two dimensions. First, it should be applied not just to the simple WHEN or VERIFY construct, but also to the WHENEVER, EVERY, and WAIT constructs. Second, it should be possible to specify the conditions for terminating the construct not just in terms of an time interval, but also in terms of a more general condition expressed as a logical expression. These capabilities are provided by the BEFORE and WITHIN clauses, as described in this section.

BEFORE and WITHIN may be used to terminate the operation of a WHEN, WHENEVER, WAIT or EVERY construct if a specified condition occurs, or a time interval elapses, before the main construct itself concludes. If such early termination occurs, the block of statements following an optional OTHERWISE clause is executed instead of the block of statements executed upon occurrence of the WHEN, WHENEVER, WAIT or EVERY condition.

Note to TIMELINER users: The BEFORE and WITHIN clauses are identical to the UNTIL and FOR clauses of the TIMELINER language, except that they include the optional capability of an OTHERWISE clause containing statements to be executed if the BEFORE or WITHIN condition occurs before the main condition. The words "before" and "within" were chosen because they read better in conjunction with "otherwise".

before_clause ::= **before** *logical*_expression

otherwise_clause ::= **otherwise**
block_of_statements

The BEFORE clause immediately follows the WHEN, WHENEVER, WAIT, or EVERY statement that it modifies.

When a BEFORE clause is used, the condition specified in the BEFORE clause is evaluated before each time the condition in the main statement is evaluated. If the BEFORE condition is satisfied, the block of statements that follow the BEFORE clause is skipped, and the block of statements that follow the OTHERWISE clause (if any) are executed instead. Then execution resumes at the statement following the END statement.

Thus, satisfaction of a BEFORE condition can be used to terminate the loop created by a WHENEVER or EVERY statement. A BEFORE clause may also be used to terminate the wait implied by a WHEN or WAIT statement. If used with WHEN, BEFORE can be used to execute different statements depending on which of two conditions arises first.

(It may be useful to allow the BEFORE statement to be used also to terminate a loop established by the FOR statement defined in section 3.5.7.3 of the document under review.)

Here is how the BEFORE clause looks when used with a WHEN, WHENEVER, WAIT or EVERY statement:

```
When condition                  Whenever condition
   Before condition                Before condition
      statements                      statements
   Otherwise                       Otherwise
      statements                      statements
End When                        End Whenever

Wait interval                   Every interval
   Before condition                Before condition
      statements                      statements
   Otherwise                       Otherwise
      statements                      statements
End Wait                        End Every
```

In each case the OTHERWISE clause and the action statements that follow it are optional. If they are not present, satisfaction of the BEFORE condition before the WHEN, WHENEVER, WAIT or EVERY condition simply causes the first block of statements to be skipped.

---

within_clause ::= **within** *timing*_expression

otherwise_clause ::= **otherwise**
block_of_statements

The WITHIN clause immediately follows the WHEN, WHENEVER, WAIT, or EVERY statement that it modifies.

When a WITHIN clause is used, the time that has elapsed since the original evaluation of the main construct is compared to the time given in the WITHIN statement. If the period of time has elapsed, the block of statements that follow the WITHIN clause is skipped, and the block of statements that follow the OTHERWISE clause (if any) are executed instead. Then execution resumes at the statement following the END statement.

Thus, a WITHIN statement can be used to establish a limit on the period during which the loop created by a WHENEVER or EVERY statement will continue to be processed, or on the period of time that the condition specified by a WHEN or WAIT statement will be looked for.

(It may be useful to allow the WITHIN statement to be used also to terminate a loop established by the FOR statement defined in section 3.5.7.3 of the document under review.)

Here is how the WITHIN clause looks when used with a WHEN, WHENEVER, WAIT or EVERY statement:

```
When condition                    Whenever condition
   Within interval                   Within interval
      statements                        statements
   Otherwise                         Otherwise
      statements                        statements
End When                          End Whenever

Wait interval                     Every interval
   Within interval                   Within interval
      statements                        statements
   Otherwise                         Otherwise
      statements                        statements
End Wait                          End Every
```

Note that the use of WITHIN with WAIT is nonsensical if literals are used to indicate the time intervals because it will be obvious which interval will expire first. However, if one or both of the intervals is specified by a variable, the construct may be useful.

In each case the OTHERWISE clause and the action statements that follow it are optional. If they are not present, expiration of the WITHIN interval before the WHEN, WHENEVER, WAIT or EVERY condition occurs simply causes the first block of statements to be skipped.

---

Note that I have placed the BEFORE and WITHIN clauses on a separate line from the statement that they modify. This decision was motivated by a desire to allow more room for compound logical expressions on the main statement line and the BEFORE line. There is no logical reason why they could not be placed on the same line as the main statement, like the WITHIN clause of the existing VERIFY statement.

## 2.6 Capabilities Added by the New Constructs

The WHEN, WHENEVER, WAIT, EVERY, BEFORE and WITHIN constructs described above can be summarized by the following table:

| | logical expression | time interval |
|---|---|---|
| one-time | WHEN | WAIT |
| establish a loop | WHENEVER | EVERY |
| terminate main construct and execute optional OTHERWISE block | BEFORE | WITHIN |

The six words WHEN, WHENEVER, WAIT, EVERY, BEFORE and WITHIN interact synergistically to provide 12 non-redundant combinations.

To evaluate the proposed new constructs two questions must be asked. First, do the "new" constructs provide the functionality of the "old" ones? Second, how does the functionality of the "new" constructs add to that already provided by the "old" ones? (The words "old" and "new" are used in this section and the next as a compact way to refer to the current UIL constructs, as compared to the alternative constructs proposed in this report.)

The ways in which the "old" UIL constructs would be implemented by the "new" constructs are indicated by examples given in Part 1. This mapping is summarized in the following table, in which "C" indicates a condition and "T" an interval of time:

| OLD | NEW |
|---|---|
| command qualifiers | WHEN, EVERY, BEFORE, WITHIN |

The capabilities of "command qualifiers" are so ambiguous that a clear mapping is difficult -- however, anything they can do can be done, without ambiguity, by simple combinations of the new constructs.

```
---------------------------------------------------------------------
Wait T                                Wait T
   statements                            statements
                                      End Wait
---------------------------------------------------------------------
Verify C Within T                     When C
   Then                                  Within T
      statements_1                          statements_1
   Otherwise                             Otherwise
      statements_2                          statements_2
End Verify                            End When
---------------------------------------------------------------------
While C                               Every T
   Repeat                                Before not C
      statements                            statements
End While                             End While
```

Note that the use of EVERY allows the repetition interval to be specified. *T* may be chosen such that EVERY interval is equal to that provided by the REPEAT clause of the WHILE statement.

```
---------------------------------------------------------------------
Handler for C is                      Whenever C
   statements                            statements
End C                                 End Whenever
```

Note that the "old" case does not apply unless *C* is a single event.

---

The ways in which the proposed "new" UIL constructs would be implemented by the "old" constructs is summarized in the following table. I have made an honest effort to find the best way to use the old constructs to accomplish each function.

```
NEW                          OLD
---------------------------------------------------------------------
When C                       Verify C Within INFINITY
   statements                   Then
End When                           statements
                             End Verify
---------------------------------------------------------------------
When C1                      Repeat
   Before C2                    Verify C2
      statements_1                 Then
   Otherwise                          statements_2
      statements_2                    Exit
End When                        End Verify
                                Verify C1
                                   Then
                                      statements_1
                                      Exit
                                End Verify
```

```
                          End Repeat
--------------------------------------------------------------------------
When C                    Verify C Within T
   Within T                  Then
      statements_1              statements_1
   Otherwise                 Otherwise
      statements_2              statements_2
End When                  End Verify
--------------------------------------------------------------------------
Whenever C                Handler for C is
   statements                statements
End Whenever              End C

                                  OR

                          Repeat
                             Verify C
                                Then
                                   statements
                             End Verify
                             Verify not C Within INFINITY
                             End Verify
                          End Repeat
```

The first version works only if *C* is an event.

```
--------------------------------------------------------------------------
Whenever C1               Repeat
   Before C2                 Verify C2
      statements_1              Then
   Otherwise                       statements_2
      statements_2                 Exit
End Whenever                 End Verify
                             Verify C1
                                Then
                                   statements_1
                                   Verify not C1 Within INFINITY
                                   End Verify
                             End Verify
                          End Repeat
```

Since events are objects, presumably the "old" version given above will work even if *C1* and *C2* are events. A different implementation using "event handlers" is possible, but it would be more complex because the *C1* event handler would require an external procedure or event handler to disable it if *C2* occurs.

```
--------------------------------------------------------------------------
Whenever C                TO = time
   Within T               Repeat
      statements_1           Verify time >= TO + T
   Otherwise                    Then
      $statements_2$               statements_2
```

```
End Whenever                       Exit
                             End Verify
                             Verify C
                                Then
                                   statements_1
                                   Verify not C Within INFINITY
                                   End Verify
                             End Verify
                          End Repeat
```

Since events are objects, presumably the "old" version given above will work even if *C* is an event. The event handler version is worse.

```
------------------------------------------------------------------------
Wait T                            Wait T
   statements                        statements
End Wait
------------------------------------------------------------------------
Wait T                            Verify C Within T
   Before C                          Then
      statements_1                      statements_2
   Otherwise                         Otherwise
      statements_2                      statements_1
End Wait                          End Verify
------------------------------------------------------------------------
Wait T1                           T0 = time
   Within T2                      Verify time >= T0 + T1 Within T2
      statements_1                   Then
   Otherwise                            statements_1
      statements_2                   Otherwise
End Wait                                statements_2
                                  End Verify
```

This case is not meaningful if *T1* and *T2* are literals. If one or both are variables it is useful for executing different statements depending on which of two time intervals is shorter.

```
------------------------------------------------------------------------
Every T                           Repeat
   statements                        statements
End Every                            Wait T
                                  End Repeat
------------------------------------------------------------------------
Every T                           Repeat
   Before C                          Verify C
      statements_1                      Then
   Otherwise                               statements_2
      statements_2                         Exit
End Every                            End Verify
                                     statements_1
                                     Wait T
                                  End Repeat
------------------------------------------------------------------------
```

```
Every T1                          TO = time
   Within T2                      Repeat
      statements_1                   Verify time >= TO + T2
   Otherwise                            Then
      statements_2                         statements_2
End Every                                  Exit
                                     End Verify
                                     statements_1
                                     Wait T1
                                  End Repeat
------------------------------------------------------------------------
```

The foregoing tables permit a comparison to be made between the straightforwardness of the existing UIL constructs and the new constructs proposed in this memo.

Of course I may be prejudiced, but my reading of the tables indicates that the new constructs are never less graceful, and often much more so, that the existing ones. The new constructs read more like plain English. I believe that if the two sets of constructs are evaluated using more complex and realistic examples, the difference will become even more apparent.

RECOMMENDATION: Add the constructs described above (WHEN, WHENEVER, WAIT, EVERY, BEFORE, WITHIN) to the User Interface Language. Take out constructs that are special cases of the new constructs: VERIFY, WHILE, event handlers, and some of the command qualifiers.

## 2.7 Comparative Examples

This section compares the "old" and "new" versions of the UIL constructs for specifying the preconditions for actions using examples similar to those used in the UIL Specification.

*Example 1: the simple WHEN...*

The present UIL provides two ways to implement the simple WHEN statement -- a statement that requests an indefinite wait until a specified condition is met. A case that would be written in the following way using the "new" constructs:

```
When TIME >= WAKE_UP_TIME or CREW ALARM
   Issue WAKE UP CALL
   Prepare COFFEE
End When
```

could be written using the VERIFY construct as:

```
Verify TIME >= WAKE_UP_TIME or CREW ALARM Within INFINITY
   Then
      Issue WAKE UP CALL
      Prepare COFFEE
   End Verify
```

or using the WHILE construct as:

```
   While TIME < WAKE_UP_TIME and not CREW ALARM
      Repeat
   End While
   Issue WAKE UP CALL
   Prepare COFFEE
```

Neither solution is horrible, but neither is as clean as it should be. In the first case the kluge-clause "Within INFINITY" must be added to convert the VERIFY construct from an immediate test to one that will wait until the condition in question occurs. In the second case the condition must be expressed as its logical complement and an empty WHILE construct is used.

The fact that there are two equally attractive ways to implement "when" may in itself suggest that the language is not optimally structured.

I believe that a concept so basic as "when" should be one of the "primitive" condition constructs of the UIL language. In other words, it should be one of the basic constructs rather than requiring a modified form of another construct. Constructs such as WHILE and VERIFY/WITHIN can then be constructed by adding a modifying clause to WHEN.

*Example 2: the WHEN/BEFORE...*

The WHEN/BEFORE statement offers a clean way of executing different actions depending upon which of two conditions occurs first. Consider the following sequence, which assumes the existence of an instrument that is turned on and then, after an indeterminate time, returns either a FAULT or a DATA_GOOD flag:

```
   Turn On INSTRUMENT
   When  INSTRUMENT DATA_GOOD
      Before  INSTRUMENT FAULT
         Call INSTRUMENT DATA_COLLECTION
      Otherwise
         Issue INSTRUMENT ALERT MESSAGE
   End When
   Turn Off INSTRUMENT
```

Using the present UIL constructs the function could be written as follows:

```
   Turn On INSTRUMENT
   Repeat
      Verify INSTRUMENT FAULT
         Then
            Issue INSTRUMENT ALERT MESSAGE
            Exit
      End Verify
      Verify INSTRUMENT DATA_GOOD
         Then
            Perform INSTRUMENT DATA_COLLECTION
            Exit
      End Verify
   End Repeat
   Turn Off INSTRUMENT
```

I believe the "new" way of expressing this function is easier to read and understand, not to speak of more concise.

*Example 3: the EVERY/WITHIN statement...*

It will often be desirable to set up a UIL sequence that will capture data helpful for error analysis whenever some off-nominal or unexplained condition occurs. Such a function will be especially useful when UIL is applied to the ground testing of SSF systems, but it may also have an application on-board.

Consider the following sequence:

```
Whenever SYSTEM GLITCH
   Every 0.05 SECONDS
      Within 1 SECONDS
         Print ANALYSIS DATA
   End Every
End Whenever
```

In other words, whenever the glitch occurs, a piece of data that changes rapidly is printed every 50 milliseconds for one second. Such a task is the bread and butter of a UIL type language when it is being used to debug a system under development.

This function could be implemented as follows using the existing constructs:

```
Repeat
   Verify GLITCH
      Then
         T_START = TIME
         Repeat
            Exit If TIME >= T_START + 1
            Print ANALYSIS DATA
            Wait 0.05 SECONDS
         End Repeat
   End Verify
End Repeat
```

Besides its excessive length, the latter implementation requires nested REPEAT constructs. Assuming that REPEAT constructs may be nested -- the UIL Spec is silent on the subject -- an ambiguity is created with respect to the EXIT statement. As written, the EXIT statement is intended to break out of the inner REPEAT loop but not the outer one.

*Example 4: the WHENEVER with limits...*

Consider the following sequence, which assumes the existence of an experiment that has a pointable optical sensor that must be protected from the sun:

```
Whenever DAY TERMINATOR CROSSING
   Whenever EXPERIMENT is ACTIVE AND LOS_ANGLE_TO_SUN < 15 DEG
      Until  NIGHT TERMINATOR CROSSING
         Issue EXPERIMENT LINE-OF-SIGHT CREW ALERT MESSAGE
         Wait 10 SECONDS
            If LOS_ANGLE_TO_SUN < 15 DEG
               Close EXPERIMENT APERTURE DOOR
            End If
         End Wait
   End Whenever
End Whenever
```

In other words: Whene<u>ver</u> the SSF passes into daylight a loop is set up that operates whenever the experiment is active <u>and</u> the line-of-sight comes too close to the Sun. Whenever this compound condition occurs a message is issued to the crew. If the hazardous condition is not remedied after 10 seconds have elapsed the sequence closes the aperture. The loop then falls through and returns to the inner WHENEVER statement, which will wait for the next off-to-on transition of the compound condition. At nightfall, the inner WHENEVER structure is terminated, and the outer WHENEVER waits again for daybreak.

To perform this function using the event handler construct would require two off-line event handlers, one of which would set a parallel procedure (in a separate environment) into motion upon crossing into daylight and the other of which would suspend it upon crossing into night. The procedure itself would contain a VERIFY or CHOICE statement within a REPEAT loop to recognize that the experiment is active and the hazardous condition exists, and within that statement would be nested the issuing of the alert message, followed by a WAIT, followed by a VERIFY/WITHIN construct whose OTHERWISE clause would close the aperture door.

If we assume that an "event" such as DAY TERMINATOR CROSSING may be detected by VERIFY the event handler construct can be avoided. The example could then be implemented as follows using existing constructs:

```
Repeat
  Verify DAY TERMINATOR CROSSING
    Then
      Repeat
        Exit If NIGHT TERMINATOR CROSSING
        Verify EXPERIMENT is ACTIVE and LOS_ANGLE_TO_SUN < 15 DEG
          Then
            Issue EXPERIMENT LINE-OF-SIGHT CREW ALERT MESSAGE
            Wait 10 SECONDS
```

```
                  Verify LOS_ANGLE_TO_SUN < 15 DEG
                    Then
                      Close EXPERIMENT APERTURE DOOR
                  End Verify
                  Verify EXPERIMENT isn't ACTIVE or LOS_ANGLE_TO_SUN >= 15 DEG
                  End Verify
              End Verify
            End Repeat
        End Verify
      End Repeat
```

As in example 3 the EXIT statement should exit the inner REPEAT loop only.

## 2.8 Desirability of the IF/ELSE Statement

In some of the examples previously quoted an IF/ELSE statement has been used. I have been informed that "the If-Then-Else battle has already been fought". However, the IF statement is so simple and well understood that it would form a valuable addition to the UIL. I hope we will reconsider its rejection.

The document under review makes the following argument against use of the IF statement (page 3-57):

> For the UIL the Verify statement is preferred over the classic IF statement because in command and control environments there are factors that can make it difficult to determine if a condition is met. For example there is latency between a command stimulus and a response via telemetry, and telemetry streams are also subject to dropout.

Obviously no IF condition can be checked before the data required to evaluate the condition is obtained, and this applies equally even if the statement is written using a construct called VERIFY or CHOICE. The UIL user will easily understand that the use of IF implies that measures will be taken to obtain valid data upon which the choice can be based.

Incidentally, the quoted argument against use of the "if" is violated by the UIL Spec itself in the case of the EXIT IF statement (section 3.5.7.4).

I believe that UIL will shoot itself in the foot if it renounces the IF/ELSE statement. Such a statement is easier to understand, and more familiar, than statements such as the CHOICE or VERIFY statements. Therefore its inclusion would favorably impact the understandability and user-friendliness of the user interface language. This benefit, to my way of thinking, far outweighs any nit-picking argument such as the one quoted above.

Therefore it is recommended that the IF/ELSE statement be included in the UIL, as follows:

```
if_statement  ::=  if logical_expression
                      block_of_statements
                   {elseif_clause}
                   [else_clause]
                   end if

elseif_clause  ::=  else if logical_expression
                       block_of_statements

else_clause  ::=  else
                     block_of_statements
```

The IF statement, along with its subsidiary ELSEIF and ELSE statements, can be used to perform an immediate check upon some condition. These statement forms function exactly as they do in a programming language except that some small amount of time may be required to assemble the data required for performing the checks.

The block of statements created by the IF, ELSEIF, and ELSE statements must be closed by an END statement. The block of statements following the IF, ELSEIF or ELSE statements may contain nested condition constructs.

RECOMMENDATION: Add the IF/ELSE construct described above to the User Interface Language.

## CONCLUSION

The UIL language described in the document under review is strong when considered as a language used by astronauts, flight controllers or system operators for entering commands that will be immediately carried out. It has significant weaknesses when considered as a language for specifying actions that must occur at some time in the future when certain conditions occur.

The document pays lip-service to the idea that the space station UIL should resemble plain English. However, many of the constructs proposed in the document are based on the format and constructs of the **Ada** programming language. If UIL were designed primarily for use by persons familiar with **Ada**, this slant might be beneficial. In fact, the UIL must also be understandable to people such as astronauts and controllers who may not be skilled in the use of any programming language, much less **Ada**.

The potential impacts of equipping the SSFP with a sub-optimal UIL include:

(1) Increased CPU and core requirements for the computer software that will "compile" UIL scripts and the software that will execute them. A more general UIL will require less elaborate software to implement.

(2) Increased core requirements for the storage of UIL scripts, both on-board and on the ground. A "cleaner" UIL will result in more compact scripts.

(3) Increased testing of scripts will be required to achieve an equal level of reliability, or equal testing will provide less reliability.

(4) Perhaps most important, a UIL that is more graceful, and easier to understand, will make it less likely that those people who will write UIL scripts during the long lifetime of the SSF program will make mistakes that may have costly effects in terms of money, operations, or safety.

In this report I have made recommendations that attempt to remedy the UIL's perceived shortcomings in the areas of capability and readability. The proposed "new" constructs need to be judged in comparison to the "old" constructs described in the UIL Specification.

I have attempted to compare the two versions using examples similar to those used as illustrations in the UIL Spec. It would be useful to undertake a more detailed comparison -- a "fly-off" -- using a longer, more realistic example supplied by a system or experiment developer. Even if there were only one version, such an evaluation should be performed before the UIL Spec is baselined

I am willing, if asked, to undertake the action of rewriting the parts of the UIL Spec that are affected by the changes I have recommended. This rewrite could occur either before or after a decision is made with regard to my recommendations. If done beforehand it would allow competing versions of the UIL Spec to be compared as alternative candidates for baselining. If the decision is to synthesize the two versions, I will be happy to assist the process in any way I can.

# User's Guide to the Timeliner Language

by

Don Eyles

December 18th, 1986

The Charles Stark Draper Laboratory, Inc.
Cambridge, Massachusetts 02139

## PREFACE

This document describes the use of TIMELINER, a program which implements a test input language designed to enable the user of a simulation conveniently to specify certain actions -- such as software loads or crew input -- and and **when** they are to occur.

The central concept of the timeline language described in this document is that the user must be able to specify actions to occur in sequence, and sequences to occur in parallel. Often a single sequential input stream is sufficient, but frequently it is not.

A particular timeline sequence may be as small, for example, as a single WHENEVER construct. Or it may be a long string of WHEN constructs. It may even be a "subsequence" invoked from another sequence. Input streams may be broken down by discipline (flight control, guidance, etc.), or by any other rule that is convenient. Most important, if two circumstances requiring action may occur in either order, the user can, by judiciously constructing two timeline sequences, achieve the result desired.

## ACKNOWLEDGEMENT

This report was prepared by The Charles Stark Draper Laboratory, Inc. under Contract NAS-48834 with the National Aeronautics and Space Administration.

Publication of this report does not constitute approval by the National Aeronautics and Space Administration of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

# TABLE OF CONTENTS

Section Page

pages=.

## 1.0 LINES AND SEQUENCES

All user input to the software on channel 5 will go to or through the timeline processor, TIMELINER. This includes the initial ILOAD input.

On its first invocation TIMELINER reads the entire channel 5 input stream, performs error checking, and then obeys those requests, such as ILOADs, which call for immediate action. Thereafter, TIMELINER will execute cyclically.

The unit of input to the timeline program is the "line". The word "statement" is used to refer to the contents of a given "line". A statement must be contained on a single line, with the single exception of the LOAD statement.

A statement is made up of one or more alphanumeric words. Words are separated from each other by one or more blanks and must not contain blanks. (The only exception being that blanks are allowed within a character string literal enclosed by single quotes.) The input-reading routine sees commas and semicolons as blanks, so the user may use these symbols at will to enhance readability.

All material after the first double quote, or exclamation point, on a line, unless it is inside a string literal, is considered a comment.

Words add up to lines and lines, in turn, add up to "sequences" (and "subseqs"). A sequence is a stream which runs in parallel with other sequences. A sequence is analogous to the MEANWHILE construct in the SLS input language. The user may specify as many or as few sequences as he requires. All input, including ILOADs, must belong to a sequence.

### 1.1 SEQUENCE HEADER

The sequence header line is of the form

```
  SEQ         <name of sequence>
    or
  SEQUENCE    <name of sequence>
```

where <name of sequence> is any alphanumeric word without imbedded blanks.

The SEQUENCE header sets up a sequence of TIMELINER statements to be processed in parallel with other sequences of TIMELINER statements. On each pass TIMELINER will examine each active sequence and execute any actions which may be called for. A sequence is deactivated when and only when it reaches its end.

The TIMELINER input stream must begin with the sequence header of the first sequence to be input. Subsequently, each new parallel sequence must begin with a sequence header. No two sequences may have the same name. No explicit sequence close statement is required. When a new sequence header or a CLOSE line is encountered during initialization, the current sequence is closed.

## 1.2 SUBSEQUENCE HEADER

The subseq header line is of the form

```
  SUBSEQ   <name of subsequence>
    or
  SUBSEQUENCE   <name of subsequence>
```

where <name of subsequence> is any alphanumeric word without imbedded blanks. A subsequence is the same as a sequence except that it does not establish an additional parallel stream. A subsequence is executed only when "CALLed" (see "2.11 CALL Statement" on page 10) from another sequence. A subsequence may be called from two or more sequences or from another subsequence.

## 1.3 CLOSE LINE

The input stream must end with a line of the form

```
  CLOSE
```

The CLOSE line terminates the input stream as a whole. The CLOSE line has no meaning at run time. It specifically does not terminate the simulation, merely the TIMELINER input stream.

## 2.0 CONTROL STATEMENTS

A given timeline sequence consists of action statements and control statements.

An action statement specifies the end to be achieved, the action desired. Control lines control the processing of a given timeline sequence so that the actions occur as required by the user. In general, a particular timeline sequence will consist of action lines (one or more together) interspersed with control lines.

Control statements which determine when actions are to occur are WHEN, WHENEVER, UNTIL, WAIT, EVERY, FOR, IF, and ELSE. These are sometimes called condition statements, because they specify conditions which must obtain before TIMELINER will move on to the ensuing actions.

Additional control statements are required for blocking and transfer of control within the timeline sequences. These are DO, END, and CALL.

### 2.1 WHEN STATEMENT

The WHEN line is of the form

```
  WHEN   <comparison>
    or
  WHEN   <comparison>  AND/OR/XOR  <comparison>
```

The entity <comparison> usually consists of three words, such as "ALT < 400000", of which the middle specifies the relationship between the first and third words which must occur before the comparison will be evaluated as true. The details of the <comparison> entity are described in "2.12 Comparison Expressions" on page 10.

A WHEN line, when encountered, causes the timeline sequence of which it is a part to pause until the condition specified is met. When the condition is achieved, TIMELINER moves on to the next line, typically an action line.

A WHEN statement with no <comparison> is also accepted as legal input. When encountered, such a WHEN statement is considered to have been satisfied, and TIMELINER will immediately move on to the next line.

The simplest useful timeline consists of WHEN statements and action statements, for example:

```
  SEQUENCE ONE
  ------------
     WHEN  TSIM > 1200  OR  ALT < 400000
        <action statement>
        <action statement>
     WHEN  TSIM > 1500
        <action statement>
        <action statement>
```

This sequence waits until either TSIM is greater than 1200 or ALT is below 400000, then executes two action statements, then waits for TSIM to be greater than 1500, before executing two more action statements. The sequence is then deactivated.

## 2.2 WHENEVER STATEMENT

The WHENEVER line is of the form

```
  WHENEVER   <comparison>
    or
  WHENEVER   <comparison>  AND/OR/XOR  <comparison>
```

A WHENEVER line when encountered causes the timeline sequence of which it is a part to pause until the condition specified is met. When the condition is achieved, TIMELINER moves on to the ensuing action statements and executes them up to (but not including) the next condition statement (WHEN, WHENEVER, WAIT, EVERY, IF) or to the end of the sequence. TIMELINER then returns to the WHENEVER line. The next time the condition is seen to become true -- i.e. on the next OFF-to-ON transition of the overall logical result -- the pattern repeats.

Note that a WHENEVER test on a time is nearly always inappropriate. For example the statement

```
  WHENEVER  TSIM > 1000
```

will only be satisfied once because TSIM will never again go from being below to being above the criterion. In this case a simple WHEN should be used.

Only if the WHENEVER line is followed by an UNTIL line (see "2.5 UNTIL Statement" on page 6) or FOR line (see "2.6 FOR Statement" on page 7 ) can the particular timeline sequence ever advance to the next condition line.

Here is a simple timeline sequence using WHENEVER:

```
  SEQUENCE TWO
  ------------
     WHENEVER  ACC  >  32
        <action statement>
        <action statement>
        <action statement>
```

This sequence executes three action statements at any time in the simulation when variable ACC goes from below 32 to above 32. This sequence stays active throughout the simulation.

## 2.3 WAIT STATEMENT

The WAIT line is of the form

```
  WAIT   <numeric>
```

where <numeric> is either a numeric literal (see "2.12.5 Numeric Literal" on page 12), a recognized numeric variable (see "2.12.2 Numeric Variable" on page 11), or an arithmetic combo (see "2.12.7 Arithmetic Combo" on page 13).

A WAIT line, when encountered, causes the timeline sequence of which it is a part to pause for the amount of time indicated. If <numeric> is a variable or arithmetic combo, it is reevaluated on every pass.

Note that since TIMELINER executes cyclically the granularity with which the desired timing occurs is the TIMELINER period. (This granularity, for the AFE simulation, is 0.050 seconds.)

Here is an example of a simple timeline sequence using WAIT:

```
  SEQUENCE THREE
  --------------
     WAIT    90
        <action statement>
        <action statement>
     WAIT    NAV_DT
        <action statement>
        <action statement>
     WAIT    1:30:17
        <action statement>
        <action statement>
```

This sequence waits 90 seconds, executes action statements, waits the time given by the variable NAV_DT, executes action statements, waits one hour and 30 minutes and 17 seconds, and executes action statements. The sequence is then deactivated.

## 2.4 EVERY STATEMENT

The EVERY statement is of the form

```
  EVERY   <numeric>
```

where <numeric> is either a numeric literal (see "2.12.5 Numeric Literal" on page 12), a recognized numeric variable (see "2.12.2 Numeric Variable" on page 11), or an arithmetic combo (see "2.12.7 Arithmetic Combo" on page 13). The EVERY statement is useful for making an action occur repetitively.

When an EVERY line is encountered for the first time TIMELINER immediately moves on to the ensuing action statements and executes them up to, but not including, the next condition statement. TIMELINER then returns to the EVERY line and pauses for the period given by the numeric literal, variable, or arithmetic combo. When this period expires, TIMELINER again executes the ensuing action statements and returns to the EVERY line, where it waits again for the period given by the literal or (reevaluated) variable.

Note that unless the EVERY line is followed by an UNTIL line or a FOR line, its action statements will be executed every <numeric> until the end of the simulation.

Here is a simple timeline using EVERY:

```
  SEQUENCE FOUR
  -------------
     EVERY   0.050
        <action statement>
        <action statement>
        <action statement>
```

This sequence executes three action statements every 50 milliseconds throughout the simulation.

## 2.5 UNTIL STATEMENT

The UNTIL statement is of the form

```
  UNTIL   <comparison>
    or
  UNTIL   <comparison>  AND/OR/XOR  <comparison>
```

The UNTIL statement is only valid when it immediately follows a WHEN, WHENEVER, WAIT or EVERY statement. Its effect is to limit how long TIMELINER will patiently wait for fulfillment of the preceeding condition statement.

When an UNTIL statement is present following another condition statement, the UNTIL statement is evaluated first. If the conditions in the UNTIL statement are passed, TIMELINER will skip over the action statements which immediately follow, and will resume processing with the next condition statement. If the UNTIL statement does not pass, the condition statement that preceeds it is evaluated.

Here is a sequence in which UNTIL statements are used to end a WHENEVER loop and to terminate a WAIT:

```
SEQUENCE FIVE
-------------
   WHENEVER    ACC  >  32
   UNTIL       MET  >  6000
      <action statement>
      <action statement>
   WAIT        10
   UNTIL       ROLLSTOP  =  ON
      <action statement>
      <action statement>
```

Until 6000 seconds is reached, this sequence executes two action statements at any time when ACC goes from below 32 to above 32, then the sequence waits 10 seconds, and then, unless ROLLSTOP first becomes equal to ON, executes two more action statements before being deactivated.

## 2.6 FOR STATEMENT

The FOR line is of the form

```
FOR   <numeric>
```

where <numeric> is either a numeric literal (see "2.12.5 Numeric Literal" on page 12), a recognized numeric variable (see "2.12.2 Numeric Variable" on page 11), or an arithmetic combo (see "2.12.7 Arithmetic Combo" on page 13). FOR bears the same relation to WAIT and EVERY as UNTIL bears to WHEN and WHENEVER. A FOR line is only valid when it follows a WHEN, WHENEVER, WAIT or EVERY statement.

When a FOR statement is present following another condition statement the FOR statement is evaluated first on each pass. If it is true, i.e. if the specified time has elapsed, the immediately following action statements are skipped and processing resumes with the next condition statement. If the FOR statement does not pass, the condition statement that preceeds it is evaluated.

Here is an example showing correct use of the FOR statement:

```
  SEQUENCE SIX
  ------------
     WHENEVER    ACC  >  32
     FOR         100
        <action statement>
        <action statement>
     EVERY       10
     FOR         320
        <action statement>
        <action statement>
```

For the first 100 seconds this sequence executes two action satements at any time that the variable ACC goes from below 32 to above 32, then for the next 320 seconds it executes two action statements every 10 seconds, before being deactivated.

## 2.7 IF STATEMENT

The IF line is of the form

```
  IF   <comparison>
    or
  IF   <comparison>  AND/OR/XOR  <comparison>
```

The IF statement allows decision making to be performed without regard to time. When an IF statement is encountered, its conditions are evaluated. If the conditions are satisfied TIMELINER resumes processing with the statement which immediately follows. If the conditions are not satisfied TIMELINER will skip to the ensuing ELSE statement if there is one, or otherwise to the next condition statement.

An example is provided in the next section.

## 2.8 ELSE STATEMENT

The ELSE statement consists of the single word "ELSE". The ELSE statement must be the next condition statement (at the same DO-END level) following an IF statement. If the IF statement fails, TIMELINER will resume processing at the ELSE statement, and the action statements which immediately follow it will be executed. If the IF statement passes, the ELSE statement and its ensuing action statements are skipped.

Here is an example showing correct use of IF-ELSE:

```
SEQUENCE SEVEN
--------------
   WHEN  T_SIM  >  6000
   IF    ALT  >  400000
      <action statement>
      <action statement>
   ELSE
      <action statement>
      <action statement>
```

When TSIM reaches 6000 this sequence will execute the first batch of action statements if ALT is above 400000, otherwise it will execute the second batch of action satements. The sequence is then deactivated.

## 2.9 DO STATEMENT

The DO statement consists of the single word "DO". It is required only for the case where condition statements are to be part of the overall action to be performed upon fulfillment of a WHENEVER or EVERY statement, or to be skipped upon fulfillment of an UNTIL or FOR statement.

When a group of condition and action statements are enclosed within a DO-END pair, those statements are all executed upon the fulfillment of a WHENEVER or an EVERY before return to the WHENEVER or EVERY line.

Similarly, if a condition statement has an accompanying UNTIL or FOR statement, then when the UNTIL or FOR is fulfilled before the preceeding condition statement, the DO-END enclosed statements will all be skipped over.

An example is provided in the next section.

## 2.10 END STATEMENT

The END statement consists of the single word "END". Its function is to terminate the grouping initiated by a DO statement.

Here is an example showing correct use of DO-END:

```
SEQUENCE EIGHT
--------------
   WHENEVER    ACC  >  32
      DO
         <action statement>
         <action statement>
      WAIT     1.5
         <action statement>
         <action statement>
      END
```

At any point in the simulation when ACC goes from below 32 to above 32 this sequence does the following: executes two action statements, pauses for 1.5 seconds, and executes two more action statements.

## 2.11 CALL STATEMENT

The CALL line is of the form

```
CALL   <name of subseq>
```

where <name of subseq> is as described in "1.1 SEQUENCE Header" on page 1. When encountered, the CALL statement will transfer control from the sequence where the CALL appears to the top of the subsequence. Upon completion of the subsequence control returns to the line after the CALL.

Note that when called a subseq behaves like a DO-END package. Thus it will all be executed upon fulfillment of an WHENEVER or an EVERY, and will all be skipped over upon fulfillment of an UNTIL or FOR.

A subseq may be called from any other seq or subseq. Because a subseq, like all TIMELINER input, is simply a script to be acted out, reentrancy is not a problem. A subseq can be called without worrying about whether the same subseq may still be in execution due to another call from another seq.

## 2.12 COMPARISON EXPRESSIONS

Comparison expressions are components of WHEN, WHENEVER, UNTIL, and IF statements. They are used to specify conditions to be looked for. A comparison expression is of the form

```
<noun>   <comparer>   <noun>
```

Where middle element <comparer> is one of the following:

```
=      ¬=      >      >=      <      <=
```

To make Fortran habitues feel more at home the following forms are also accepted:

.EQ.   .NE.   .GT.   .GE.   .LT.   .LE.

Each side element <noun> may be either a boolean variable, an numeric variable, a character string variable, a boolean literal, a numeric literal, a character string literal, or an arithmetic combo. The nouns on each side of the <comparer> must be compatible. The various types of noun are described below.

### 2.12.1 Boolean Variable

A boolean variable is any unarrayed boolean recognized by TIMELINER. A boolean variable is suitable for "=" or "¬=" comparison with a boolean literal or a boolean variable, and with a character string variable if the latter converts sucessfully to boolean form.

A boolean variable may be modified by the symbol "¬" which must be typed adjacent to the variable name without a space between. The effect of the modification is to change the polarity of the boolean variable.

Note that specific arrangements must be made within the TIMELINER program to enable it to recognize a particular Fortran variable by name. (See "5.2 Adding TIMELINER Variables" on page 24.)

Here are examples of comparison expressions involving boolean variables:

```
IGN_CMD  =  .ON.
IGN_CMD  =  ¬CUTOFF_CMD
JET_ON(3)  =  .FALSE.
```

The words ".ON." and ".FALSE." are boolean literals as described below.

### 2.12.2 Numeric Variable

A numeric variable is any unarrayed integer or scalar variable recognized by TIMELINER. A numeric variable is suitable for any sort of comparison with a numeric literal, another numeric variable, or an arithmetic combo, and with a character string variable when the latter converts sucessfully to a number.

Besides its role in comparison expressions a numeric variable may be the operand of a WAIT, EVERY or FOR statement, or of a LOAD FROM statement.

An numeric variable may be modified by the minus symbol "-" which must be typed adjacent to the variable name without a space between. The effect of the modification is to change the sign of the numeric variable.

Note that specific arrangements must be made within the TIMELINER program to enable it to recognize a particular Fortran variable by name. (See "5.2 Adding TIMELINER Variables" on page 24.)

Here are examples of comparison expressions involving numeric variables:

```
  ALT  <  400000
  TSIM  >=  T_CUTOFF
  -TGO  <  100
```

The words "400000" and "100" are numeric literals as described below.

### 2.12.3 Character String Variable

A string variable is any character or (multiple) bit string recognized by TIMELINER. A string variable is suitable for "=" and "¬=" comparison with a character string literal or another character string variable. When it converts sucessfully to a boolean, it may be compared to a boolean variable or a boolean literal. When it converts sucessfully to a numeric a character string variable may be compared in any way with a numeric literal, a numeric variable, an arithmetic combo, or another character string variable when the latter converts properly to numeric form.

Note that specific arrangements must be made within the TIMELINER program to enable it to recognize a particular Fortran variable by name. (See "5.2 Adding TIMELINER Variables" on page 24.)

Here are examples of comparison expressions involving string variables:

```
  DISPLAY_TITLE   ¬=   'ORBIT COAST'
  SPEC_PAGE(9)    =   'AUTO'
```

The words bounded by single quotes are character string literals as described below.

### 2.12.4 Boolean Literal

A boolean literal is of the form ".ON." or ".TRUE." or ".OFF." or ".FALSE.". (Note that "ON", "TRUE", "OFF" and "FALSE" are treated as numeric literals meaning 0.0, 0.0, 1.0 and 1.0 respectively.) A boolean literal is suitable for "=" or "¬=" comparison with a boolean variable, and with a character string variable if the latter converts sucessfully to boolean form.

### 2.12.5 Numeric Literal

A numeric literal is any series of characters which give an integer or scalar number. A numeric literal is suitable for comparison with a numeric variable or an arithmetic combo, and with a character string variable when the latter converts sucessfully to a numeric value.

Besides its role in comparison expressions a numeric literal may be the operand of a WAIT, EVERY or FOR statement, or of a LOAD FROM statement.

Note that because integer variables are sometimes used as binary switches the forms "ON", "TRUE", "OFF" and "FALSE" are interpreted as numeric literals with the values 0.0, 0.0, 1.0 and 1.0 respectively. (The forms ".ON.", ".TRUE.", ".OFF." and ".FALSE." are treated as boolean literals.)

Note also that while in general a numeric literal must be a valid FORTRAN arithmetic literal, the days-hours-minutes-seconds form

```
  DDD/HH:MM:SS.SS
```

and also the forms

```
  HH:MM:SS.SS   and   MM:SS.SS
```

are specifically recognized as numeric literals and for use are simply converted to seconds. For example the word "100/02:01:02.2" would be interpreted identically to "8647262.2".

### 2.12.6 Character String Literal

A string literal is any alphanumeric series at all enclosed by single quotes, and may contain blanks. A string literal is suitable for "=" or "¬=" comparison with a string variable.

### 2.12.7 Arithmetic Combo

An arithmetic combo is of the form

```
  <noun>  <operator>  <noun>
```

where the middle element <operator> is one of the following:

```
  +     -     *     /     **     MOD
```

meaning respectively, addition, subtraction, multiplication, division, exponentiation, and the modulo operation. Each side element <noun> must be either a numeric variable or a numeric literal. The arithmetic combo must be written with a space on either side of <operator>.

In a WHEN, WHENEVER, UNTIL or IF statement an arithmetic combo is suitable for any sort of comparison with a numeric literal, a numeric variable, another arithmetic combo, or a character string variable when the latter converts sucessfully to a number.

An arithmetic combo may also be the operand of a WAIT, EVERY or FOR statement, or of a LOAD FROM statement.

In addition, TIMELINER recognizes a special arithmetic combo of the form

```
  <function>  OF  <noun>
```

where <function> may be one of the following:

```
  ABS   ROUND   SQRT   SIN   COS   TAN   ARCSIN   ARCCOS   ARCTAN
```

This combo is evaluated in the obvious way. The trigonometric functions operate in radians.

Note that TIMELINER does not protect against division by zero or against out-of-range function inputs. Runtime errors will occur in these cases.

Here are some examples of comparison expressions involving arithmetic combos:

```
  T_GMT  >=  TIG - 10
  MSC_PHS  =  SIM_PASS MOD MSC_CNT
  S_SPARE(1) / S_SPARE(2)  <  S_SPARE(4) ** 2
  S_SPARE(4)  >=  COS OF THETA
```

## 2.13 VARIABLE SUBSCRIPTING

TIMELINER permits variables to be subscripted with up to three subscripts. Subscripts are given within parentheses and if more than one subscript is required they are separated by commas. Blanks are illegal between the variable and the subscript, and blanks must not occur within the subscript itself.

For use in comparison expressions and in LOAD-FROM statements, array variables must be fully subscripted. That is, all the subscripts required must be explicitly given so that the effect is to specify a single boolean, numeric, or string.

For use in LOAD and PRINT statements, an asterisk "*" may be used as a wild card subscript. The number of pieces of data required by a LOAD statement, or printed by a PRINT statement, will depend on the dimensions represented by the wild card subscript.

Suppose that the array variables FLAGS and INERTIAS were declared by the following statements:

```
  LOGICAL  FLAGS(4)
  REAL*8   INERTIAS(4,3,3)
```

In that case, the following are legal condition statements:

```
  WHENEVER  FLAG(1)  =  .ON.  &  FLAG(2)  =  .OFF.
  WHEN      INERTIAS(2,1,1)  >  0.25
```

The following are legal LOAD-FROM statements:

```
LOAD      FLAG(1)  FROM  FLAG(2)
LOAD      INERTIAS(3,1,1)  FROM  INERTIAS(2,1,1)
```

And the following are legal LOAD statements:

```
LOAD      FLAG(3)  .OFF.
LOAD      FLAG(*)  .ON.  .ON.  .ON.  .ON.
LOAD      FLAG  .OFF.  .OFF.  .OFF.  .OFF.
LOAD      INERTIAS(1,1,1)  0.1
LOAD      INERTIAS(*,1,1)  0  0  0  0
LOAD      INERTIAS(2,*,*)  1  0  0  0  1  0  0  0  1
LOAD      INERTIAS(*,*,*)  1  2  3  4  1  2  3  4  1  2  3  4
                           1  2  3  4  1  2  3  4  1  2  3  4
                           1  2  3  4  1  2  3  4  1  2  3  4
```

The immediately preceeding LOAD statements would also be legal PRINT statements if the word "PRINT" were substituted for "LOAD" and the data given after the variable were omitted.

Note that if an array variable is given without any subscripts TIMELINER assumes that the entire variable is intended, just as though all subscripts were "wild cards". For example "INERTIAS" is equivalent to "INERTIAS(*,*,*)". Both forms are legal and in run-time printout both will be printed as simply "INERTIAS".

TIMELINER will issue cusses at initialization time for any variables which are improperly subscripted.

## 3.0 ACTION STATEMENTS

The following action statements are currently implemented: ABORT, which terminates the simulation in progress; PRINT, used to print any recognized variable; LOAD, used to load (set) any recognized variable; DUMP, used to print the contents of an entire common area; and EXECUTE, used to invoke a recognized Fortran procedure.

### 3.1 ABORT STATEMENT

The ABORT statement consists of the single word "ABORT". When executed this statement sets an event which immediately causes termination of the simulation.

### 3.2 LOAD STATEMENT

The LOAD statement is of the form

```
  LOAD  <name of variable>  <data>  <data>  ...  <data>
     or
  LOAD  <name of variable>  <scale factor>  <data>  <data>  ...  <data>
```

where <name of variable> is a Fortran name which TIMELINER has been enabled to recognize, and <data> represents boolean, numeric or string type data, for example three scalars if the parameter in question were a vector. The optional modifier <scale factor>, a word such as "M_TO_FT" or "DEG_TO_RAD", is used to modify the data that is provided before it is used.

When executed the LOAD statement loads the data given, the amount of which must correspond to the quantity needed, into the specified parameter. The parameter's FORTRAN name is printed at the time the LOAD takes place, along with the value to which it is loaded.

Unlike any other kind of statement, a LOAD statement may be continued on suceeding lines. If the data provided on the line with the word LOAD and the parameter specification is not sufficient to fill up the variable, a new line is read. Unless this line is obviously of some other type, in which case a complaint is issued, its contents are interpreted as data to be loaded into the parameter specified.

Note that in some cases integer variables are used as boolean switches within a simulation. To facilitate LOADs when this is the case the forms "ON", "T", and "TRUE" are interpreted as the integer 1, and the forms "OFF", "F", and "FALSE" are interpreted as zero. Note that when the variable in question is a genuine boolean (i.e. logical) the forms

".ON." or ".T." or ".TRUE." and ".OFF." or ".F." or ".FALSE." must be used instead.

Note that specific arrangements must be made within the TIMELINER program to enable it to recognize a particular Fortran variable by name. (See "5.2 Adding TIMELINER Variables" on page 24.)

TIMELINER also recognizes a variant of the LOAD statement of the form

```
  LOAD  <name of variable>  FROM  <noun>
```

where <name of variable> must name an **unarrayed** boolean, numeric, or string. If the variable is a boolean, <noun> must be a boolean variable or literal. If the variable is a numeric, <noun> must be a numeric variable or literal, or an arithmetic combo. If the variable is a string, <noun> must be a string variable or literal. The modifier <scale factor> is illegal in a LOAD-FROM statement.

The LOAD-FROM statement is the equivalent of an assignment statement. Note that the LOAD-FROM capability, coupled with a <noun> of the arithmetic combo type, can be used to perform algebraic computations in TIMELINER language. Spare scalars or integers may be used as accumulators when the computation is complex. Although this capability is somewhat tortuous, it may occasinoally prove useful in making a run work without having to make code changes.

### 3.3 PRINT STATEMENT

The PRINT statement is of the form

```
  PRINT <name of variable>
```

where <name of variable> is a FORTRAN name which TIMELINER recognizes as a LOAD/PRINT variable.

The PRINT statement causes the variable specified to be printed. Note that specific arrangements must be made within the TIMELINER program to enable it to recognize a particular Fortran variable by name. (See "5.2 Adding TIMELINER Variables" on page 24.)

### 3.4 DUMP STATEMENT

The DUMP statement is of the form

```
  DUMP  <name of common>
     or
  DUMP  ALL
```

where <name of common> is a character string, without quotation marks, which is the name, or partial name, of a Fortran common area.

The DUMP statement, when encountered, causes all the TIMELINER variables in the specified common area to be printed (just as they would be by PRINT statements) in alphabetical order.

TIMELINER compares the <name of common> provided by the DUMP statement to the common area names that were fed into the off-line program TLPREP at the time that the variable-handling subroutine TLVARS was created. (See "5.2 Adding TIMELINER Variables" on page 24 for an explanation of this procedure.) If the string provided **is present within** the common area specification, all the variables within that common area will be dumped. Let us say that the following common areas were provided to the TLPREP program:

```
SIMCOM.FOR
ENVCOM1.CMN
ENVCOM2.CMN
```

In this case the statement:

| | |
|---|---|
| **DUMP SIMCOM** | will dump the variables in SIMCOM.FOR |
| **DUMP ENVCOM1** | will dump the variables in ENVCOM1.CMN |
| **DUMP ENV** | will dump the variables in both ENVCOM1.CMN and ENVCOM2.CMN |
| **DUMP COM** | will dump the variables in all three common areas |
| **DUMP ALL** | will dump the variables in all three common areas |

Note that when more than one common area is dumped, the order is the order in which the common areas were fed into TLPREP. For a complete description of the procedure for incorporating specific variables into TIMELINER, see "5.2 Adding TIMELINER Variables" on page 24.

## 3.5 EXECUTE STATEMENT

The EXECUTE statement is of the form

```
EXECUTE <subroutine>
```

where <subroutine> is any Fortran subroutine which TIMELINER has been enabled to recognize.

The EXECUTE statement, when encountered, causes the specified subroutine to be invoked by means of a Fortran CALL. Executable subroutines invoked by the EXECUTE statement are not allowed to have argument lists.

Explicit arrangements must be made for TIMELINER to be able to execute a particular procedure. See "5.3 Adding Executable Subroutines" on page 25 to learn how to add executable procedures.

## 4.0 TIMELINER PRINTING

A statement input to TIMELINER is, in general, printed twice. It is first printed at the start of the run when it is input, along with any error messages which may be called for. It is printed again during the run when, and if, it is executed.

### 4.1 INPUT-TIME PRINTING

At input time, a statement is echoed just as it is input. SEQ and SUBSEQ and CLOSE statements are underlined. Line numbers are provided. Spacing is regularized and indentation is applied in such a way as to make blocking more obvious. Otherwise there are no changes.

Error checking is performed at this time, and "cusses" are printed if errors are detected. Cusses are issued when the input is known not to be executable. In certain cases, the benefit of the doubt is given. For example, when a character string variable is to be magnitude compared with a scalar, no cuss is issued. If at run time the string does not convert to a scalar the comparison simply will not pass.

There is only one level of error -- fatal. The run will not be allowed to proceed if any errors are detected. However, input-time processing does continue. TIMELINER tries to find all errors the first time through. (If you have a case where it does not, I would like to see it.) A line continues to be processed even after its first cuss, and there is no maximum number of cusses at which TIMELINER will throw up its hands in dismay and quit.

There are cases where multiple complaints will stem from what is essentially one error. There may or may not be cases where spurious cusses result from a previous real error. (If you experience one, please let me know.)

Following input-time printing, a summary is provided of the file space required to tabulate the input. If users find themselves close to 100% in any catagory the file arrays should be enlarged, as described in "5.1 Increasing TIMELINER Array Sizes" on page 23.

### 4.2 RUN-TIME PRINTING

TIMELINER's method of operation is this: At input time statements are sucked dry of information and that information is stored in tables. At run time the tables are processed.

The printing that occurs at run time is a recomposition, from the tables, of the original statements. Standard forms are used so for example "&" in the input statement will be printed as "AND" at run time. Any numeric literal, even an integer, will be printed as a scalar double at run time.

Functionally, the statement should be as it was input. This provides a check of TIMELINER's tabulation of the input. If the run-time printout does not correspond to the input-time printout a serious error in the TIMELINER code is indicated.

A statement is printed at run time when and only when it is executed. (If the statement is executed more than once, it is of course printed each time.) A condition statement is printed only when it passes. As a result, in the case of a WHEN followed by an UNTIL, if the UNTIL passes before the WHEN, the WHEN will never be printed, but the UNTIL will be. Similarly, in the case of an IF-ELSE, if the IF fails, the ELSE will be printed. The user may have to refer back to the input-time printout to see the IF conditions that led, by failing, to the execution of the ELSE.

Line numbers are provided with each line printed at run time to facilitate referring back to the input printout, when necessary.

TIMELINER's run-time printout is provided in packets. Each packet contains the printing for a given sequence at a given time step. If the sequence does not advance, no printing occurs. Each packet carries a header giving the sequence name and the simulation time.

# 5.0 TIMELINER MAINTENANCE

This section describes the maintenance function which must be provided to keep TIMELINER current when it is imbedded in a simulation which is undergoing change.

## 5.1 INCREASING TIMELINER ARRAY SIZES

TIMELINER functions by analyzing the input stream at the start of the run and storing the information derived in a form suitable for processing during run time. The storage areas used for this purpose have been initialized to be quite large. Nevertheless it may occasionally be necessary to increase them.

The file usage of a particular TIMELINER input stream is summarized in the TIMELINER output. The following is an example of this summary, based on an actual run containing simple TIMELINER input:

```
FILE USAGE SUMMARY:
-------------------

                                          USED      MAX
   NUMBER OF SEQUENCES                     10       128
   NUMBER OF LINES                         58       512
   NUMBER OF NOUNS                         12       512
   NUMBER OF BOOLEAN LITERALS               0       512
   NUMBER OF NUMERIC LITERALS              51      4096
   NUMBER OF STRING  LITERALS               0      1024
```

If and when the numbers in the "USED" column grow to a magnitude approaching the numbers in the "MAX" column, it would be wise to increase the size of the appropriate file.

File sizes are controlled by parameters found in the common area TLCOM, as described in the following table:

| | |
|---|---|
| NS | number of sequences (and subseqs) |
| NL | number of lines (total for all seqs and subseqs) |
| NN | number of nouns |
| NB | number of boolean (logical) literals |
| NM | number of numeric (integer or real) literals |
| NC | number of character string literals |

To modify the file size, simply change the PARAMETER statement as required and recompile TL and TLVARS.

There is an additional parameter of interest which is not covered by the "file usage summary". This parameter, named DL, specifies the maximum depth of DO-END nesting and is initially set to 4. The depth of nesting required depends on the number of nested DO-END combinations, and on the number of nested subsequence CALLs. It may be necessary to increase DL as the TIMELINER input becomes more complex.

## 5.2 ADDING TIMELINER VARIABLES

TIMELINER is not magic. It has no way of knowing about the variables in the simulation unless it is told about them. This section describes procedures developed to facilitate this process.

The various TIMELINER functions which involve actual simulation variables are centralized in a Fortran subroutine called TLVARS. TLVARS contains four entry points, as follows:

**FIND_VAR** called when the input stream in analysed to locate the individual variables mentioned in the input

**GRAB_VAR** called during the run to retrieve the value of a variable for use in a comparison or a LOAD-FROM statement

**LOAD_VAR** called during the run to load a variable from input data

**PRINT_VAR** called during the run to print a variable

An off-line Fortran program, called TLPREP, exists for the purpose of writing the Fortran subroutine TLVARS.

Before use TLPREP must of course be compiled and linked. It is then invoked by keying

```
RUN TLPREP
```

The program TLPREP will respond by asking for common areas to be specified. It will keep asking for another common area until the question is answered by a simple carriage return.

The input that is required will depend on how the common areas in the simulation are stored. In the simplest case, where common area XCOM exists in file XCOM.CMN the required input is "XCOM.FOR". In any case the character string input must be suitable for use in a Fortran OPEN statement, and the result, if the character string is wrong, will be an OPEN failure which will occur after the last common area is input.

Once all the common areas have been input, TLPREP will read these common areas and incorporate all the variables it finds there into the subroutine TLVARS.

TLPREP leaves behind two pieces of output:

| | |
|---|---|
| **TLPREP.DAT** | which contains TLPREP output including a list of the variables which were incorporated and their characteristics, and |
| **TLVARS.FOR** | which contains the TLVARS subroutine which must then be compiled and incorporated, with TIMELINER, into the simulation link. |

In certain cases it may be undesirable to incorporate all the variables in a common area into TLVARS. For this purpose, TLPREP recognizes two directives which must appear in Fortran comment lines starting with a "C". If a comment line contains "TIMELINER_OFF" TLPREP will cease incorporating variables until it encounters a comment line which contains "TIMELINER_ON". These directives may be used any number of times to turn on and off the scan conducted by TLPREP.

Note that TLPREP will fail if it finds more than one variable with the same name in the common areas it surveys. In this case the problem may first be noticed when the subroutine TLVARS will not compile because of duplicate variable declarations.

Note one further important point: Frequently Fortran "parameters" are established in a common area which are then used to specify the dimensions of arrays, for example:

```
INTEGER  SIZE
PARAMETER (SIZE = 10)
INTEGER  ARRAYONE(SIZE)
INTEGER  ARRAYTWO(SIZE)
```

TLPREP has been written in such a way that it can cope with this method as long as the parameter statement occurs before any statement where the parameter is used. **However** TLPREP will choke if it encounters statements where a parameter is used in an arithmetic expression, as for example in the following statements:

```
PARAMETER (SIZE = OLDSIZE + 10)
   or
INTEGER  ARRAY(SIZE / 2)
```

Common areas known to TIMELINER must be written in such a way as to avoid such constructions or else they must be bypassed by the use of the "TIMELINER_OFF" and "TIMELINER_ON" directives described above.

## 5.3 ADDING EXECUTABLE SUBROUTINES

TIMELINER, by means of the EXECUTE statement, has the ability to call an arbitrary Fortran subroutine. Such subroutines must be known to TIMEL-

INER in advance and incorporated into the TIMELINER subroutine TLSUBROS which is stored in the same file as TIMELINER itself.

TLSUBROS contains two entries, FIND_SUBRO and CALL_SUBRO. To add a new subroutine changes must be made under each entry.

Under FIND_SUBRO another Fortran "ELSE IF" statement must be added to set output variable I to a number which will correspond to the subroutine to be added.

Under CALL_SUBRO the Fortran computed "GOTO" satement must be modified to execute a "CALL" statement for the given subroutine when the input variable I is equal to the value used under FIND_SUBRO.

Fortran subroutines called by CALL_SUBRO are not permitted to have argument lists.

```
           DDDDDDDDD     EEEEEEEEEEEE  EEEEEEEEEEEE      11             444           444          11      PPPPPPPPPPP
          DDDDDDDDDD    EEEEEEEEEEEE  EEEEEEEEEEEE     111            4444          4444         111      PPPPPPPPPPPP
         DD       DD   EE            EE               1111          44 44         44 44        1111      PP        PP
        DD        DD  EE            EE                 11         44  44        44  44          11      PP        PP
       DD        DD  EE            EE                 11        44   44       44   44          11      PP        PP
      DD        DD  EEEEEEEE      EEEEEEEE          11        44444444444   44444444444       11      PPPPPPPPPPPP
     DD        DD  EEEEEEEE      EEEEEEEE          11       444444444444  444444444444       11      PPPPPPPPPPP
    DD        DD  EE            EE                11              44            44          11      PP
   DD        DD  EE            EE                11              44            44          11      PP
  DD        DD  EE            EE                11              44            44          11      PP
 DDDDDDDDDD    EEEEEEEEEEEE  EEEEEEEEEEEE   1111111111         44            44      1111111111  PP
DDDDDDDDD     EEEEEEEEEEEE  EEEEEEEEEEEE   1111111111         44            44      1111111111  PP

              JJJJJJJJJ    6666666666   2222222222         444      2222222222                XX        XX
              JJJJJJJJJ   666666666666 222222222222       4444     222222222222               XX        XX
                  JJ      66       66  22        22      44 44     22        22                XX      XX
                  JJ      66                     22     44  44               22                 XX    XX
                  JJ      66                     22    44   44               22                  XX  XX
                  JJ      66666666666           22    44444444444           22                    XXXX
                  JJ      666666666666        22     444444444444         22                      XXXX
                  JJ      66       66       22              44          22                       XX  XX
            JJ    JJ      66       66     22                44        22                        XX    XX
            JJ    JJ      66       66   22                  44      22                         XX      XX
            JJJJJJJJ      666666666666 222222222222         44     222222222222               XX        XX
             JJJJJJ        6666666666  222222222222         44     222222222222               XX        XX

*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
*X  END  JOB 6242 DEE1441P 1      001 001 MVS     DEE1441        ROOM UP4B   4.11.33 PM 18 DEC 86 PRINTR8  CSDL  END  X*
```

```
            DDDDDDDDD     EEEEEEEEEEE  EEEEEEEEEEE        11            444          444          11       PPPPPPPPPPP
           DDDDDDDDDD    EEEEEEEEEEE  EEEEEEEEEEE       111           4444         4444         111       PPPPPPPPPPPP
          DD       DD   EE            EE              1111         44 44        44 44        1111       PP         PP
         DD       DD   EE            EE                11        44  44       44  44          11       PP         PP
        DD       DD   EE            EE                11       44   44      44   44          11       PP         PP
       DD       DD   EEEEEEEE      EEEEEEEE          11      44444444444  44444444444      11       PPPPPPPPPPPP
      DD       DD   EEEEEEEE      EEEEEEEE          11     444444444444 444444444444      11       PPPPPPPPPPP
     DD       DD   EE            EE                11           44           44          11       PP
    DD       DD   EE            EE                11           44           44          11       PP
   DD      DD    EE            EE                11           44           44          11       PP
  DDDDDDDDDD    EEEEEEEEEEE  EEEEEEEEEEE    1111111111       44           44      1111111111   PP
 DDDDDDDDD     EEEEEEEEEEE  EEEEEEEEEEE    1111111111       44           44      1111111111   PP


                JJJJJJJJJJ   6666666666        11       555555555555   3333333333               XX         XX
                JJJJJJJJJJ  666666666666      111       555555555555  333333333333              XX         XX
                    JJ      66        66     1111       55            33        33               XX       XX
                    JJ      66                 11       55                      33                XX     XX
                    JJ      66                 11       55                      33                 XX   XX
                    JJ      66666666666        11       555555555            3333                   XXXX
                    JJ      666666666666       11       5555555555           3333                   XXXX
                    JJ      66        66       11                55             33                 XX   XX
              JJ    JJ      66        66       11                 55            33                XX     XX
              JJ    JJ      66        66       11                 55  33        33               XX       XX
              JJJJJJJJ      666666666666   1111111111   555555555555  333333333333              XX         XX
               JJJJJJ        6666666666    1111111111   55555555555    3333333333               XX         XX


*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
*X START JOB 6153 DEE1441P 1        001 001 MVS      DEE1441             ROOM UP4B   4.10.28 PM 18 DEC 86 PRINTR8  CSDL START X*
```

# USER'S GUIDE TO THE TIMELINE LANGUAGE

# INTRODUCTION

This document describes a "timeline language" developed for use in Shuttle simulations.

The language, and this document, will undergo a process of evolution. Comments are avidly solicited. What inconveniences do you anticipate? What capabilities do you crave?

By "timeline language" is meant a test input language designed to enable the user of a simulation conveniently to specify certain actions -- such as software loads or crew input -- and when they are to occur.

We have not, in the past, had a unified way to do this. We have used SLS capability to input switch and pushbutton manipulations, PADANDCREW_LOAD routines to perform software loads, and an independent keystroke input stream for astronaut keyboard inputs.

We have never suceeded in eliminating any of these three input streams because a degree of parallelism is required in our test input. We may not know whether the circumstances calling for a keystroke input will occur before or after circumstances calling for a switch input, for example. If the order is not known, a single input timeline cannot handle both actions.

In addition, actions which should occur whenever some circumstance arises are difficult to integrate into a single input stream which must do other things as well.

Accordingly, the central concept of the timeline language here described is that the user be able to construct parallel timeline sequences, as many as may be necessary. If a single stream is sufficient, excellent. But the timeline language must provide for the case in which it is not sufficient.

A particular timeline sequence may be as small, for example, as a single WHENEVER construct. Or it may be a long string of WHEN constructs. It may even be a "subsequence" invoked from another sequence. Input streams may be broken down by discipline (flight control, guidance, etc.), or by any other rule that is convenient. Most important, if two circumstances requiring action may occur in either order, the user can, by judiciously constructing two timeline sequences, achieve the result desired.

## 1.0 LINES AND SEQUENCES

All user input to the software on channel 5 will go to or through the timeline processor, TIMELINER. This includes the initial ILOAD input.

On its first invocation TIMELINER reads the entire channel 5 input stream, performs error checking, and then obeys those requests, such as ILOADs, which call for immediate action. Thereafter, TIMELINER will execute cyclically.

The unit of input to the timeline program is the "line". The word "statement" is used to refer to the contents of a given "line". A statement must be contained on a single line.

A statement is made up of one or more alphanumeric words. Words are separated from each other by one or more blanks and must not contain blanks. The only exception is that blanks are allowed within a character/bit literal (see section 2.10.6) enclosed by single quotes. The card-reading routine sees commas and semicolons as blanks, so the user may use these symbols at will to enhance readability.

All material after the first double quote or the first asterisk on a line, provided the double quote or asterisk is not inside a character/bit literal, is considered a comment.

Words add up to lines and lines, in turn, add up to "sequences" (and "subseqs"). A sequence is a stream which runs in parallel with other sequences. A sequence is analogous to the MEANWHILE construct in the SLS input language. The user may specify as many or as few sequences as he requires. All input, including ILOADs, must belong to a sequence.

## 1.1 SEQUENCE Header

The sequence header line is of the form

SEQ <name of sequence>
or
SEQUENCE <name of sequence>

where <name of sequence> is any alphanumeric word without imbedded blanks. The channel 5 input stream must begin with the sequence header of the first sequence to be input. Subsequently, each new parallel sequence must begin with a sequence header. No two sequences may have the same name. No explicit sequence close statement is required. When a new sequence header or a CLOSE line (see 1.3) is encountered, the current sequence is closed.

## 1.2 SUBSEQUENCE Header

The subseq header line is of the form

SUBSEQ <name of subsequence>
or
SUBSEQUENCE <name of subsequence>

where <name of subsequence> is any alphanumeric word without imbedded blanks. A subsequence is the same as a sequence except that it does not establish an additional parallel stream. A subsequence is executed only when "called" (see 2.9) from another sequence. A subsequence may be called from two or more sequences or from another subsequence.

## 1.3 CLOSE Line

The input stream must end with a line of the form

CLOSE

The CLOSE line terminates the input stream as a whole. The CLOSE line is not has no meaning at run time. It specifically does not terminate the simulation.

## 2.0 CONTROL STATEMENTS

A given timeline sequence consists of action statements and control statements.

An action statement specifies the end to be achieved, the action desired. Control lines control the processing of a given timeline sequence so that the actions occur as required by the user. In general, a particular timeline sequence will consist of action lines (one or more together) interspersed with control lines.

Control statements which determine when actions are to occur are WHEN, WHENEVER, UNTIL, WAIT, EVERY, and FOR. These are sometimes called condition statements, because they specify conditions which must obtain before TIMELINER will move on to the ensuing actions.

Additional control statements are required for blocking and transfer of control within the timeline sequences. These are DO, END, and CALL.

## 2.1 WHEN Statement

The WHEN line is of the form

```
                    WHEN   <comparison>
                              or
      WHEN   <comparison>  AND/OR/XOR  <comparison>
```

The entity <comparison> usually consists of three words, such as "ALT < 100000", of which the middle specifies the relationship between the first and third words which must occur before the comparison will be evaluated as true. The details of the <comparison> entity are described in section 2.10.

A WHEN line, when encountered, causes the timeline sequence of which it is a part to pause until the condition specified is met. When the condition is achieved, TIMELINER moves on to the next line, typically an action line.

A WHEN statement with no <comparison> is accepted as legal input. When encountered, such a WHEN statement is considered to have been satisfied, and TIMELINER will immediately move on to the next line.

The simplest useful timeline consists of WHEN statements and action statements, for example:

```
SEQUENCE ONE

   WHEN  MET > 1200.0  OR  ALT < 400000

      <action statement>

      <action statement>

   WHEN  MET > 1500.0

      <action statement>

      <action statement>
```

## 2.2 WHENEVER Statement

The WHENEVER line is of the form

```
                    WHENEVER   <comparison>
                                    or
   WHENEVER   <comparison>  AND/OR/XOR  <comparison>
```

A WHENEVER line when encountered causes the timeline sequence of which it is a part to pause until the condition specified is met. When the condition is achieved, TIMELINER moves on to the ensuing action statements and executes them up to the next condition statement (WHEN, WHENEVER, WAIT, EVERY) or to the end of the sequence. TIMELINER then returns to the WHENEVER line. The next time the condition is seen to become true -- i.e. on the next OFF-to-ON transition -- the pattern repeats.

A WHENEVER statement with no <comparison> is accepted as legal input. When encountered, such a WHENEVER statement is considered to have been satisfied, and TIMELINER will act accordingly.

Only if the WHENEVER line is followed by an UNTIL (see 2.5) or FOR (see 2.6) line can the particular timeline sequence ever advance to the next condition line.

Here is a simple timeline using WHENEVER:

```
SEQUENCE TWO

   WHENEVER     ACC  >  32

       <action statement>

       <action statement>

       <action statement>
```

## 2.3 WAIT Statement

The WAIT line is of the form

```
WAIT   <numeric>
```

where <numeric> is either a numeric literal (see section 2.10.5) or a recognized integer/scalar variable (section 2.10.2).

A WAIT line, when encountered, causes the timeline sequence of which it is a part to pause for the amount of time indicated. If <numeric> is an integer/scalar variable, it is reevaluated every TIMELINER pass so volatile numbers such as a TGO may be used for delta-timing.

Note that since TIMELINER executes cyclically the granularity with which the desired timing occurs is the TIMELINER period, typically 80 ms.

Here is an example of a simple timeline sequence using WAIT:

```
SEQUENCE THREE

   WAIT     2

      <action statement>

      <action statement>

   WAIT     NAV_EXEC_DT

      <action statement>

      <action statement>

   WAIT     1:30:17.7

      <action statement>

      <action statement>
```

## 2.4 EVERY Statement

The EVERY statement is of the form

```
EVERY   <numeric>
```

where <numeric> is either a numeric literal or an integer/scalar variable. The EVERY statement is useful for making an action occur repetitively.

When an EVERY line is encountered for the first time TIMELINER immediately moves on to the ensuing action statements and executes them up to, but not including, the next condition statement. TIMELINER then returns to the EVERY line and pauses for the period given by the numeric literal or the integer/scalar variable. When this period expires, TIMELINER again executes the ensuing action statements and returns to the EVERY line, where it waits again for the period given by the literal or (reevaluated) variable.

Note that unless the EVERY line is followed by an UNTIL line or a FOR line, its action statements will be executed every <numeric> until the end of the simulation.

Here is a simple timeline using EVERY:

```
SEQUENCE FOUR

    EVERY    .96

        <action statement>

        <action statement>

        <action statement>
```

## 2.5 UNTIL Statement

The UNTIL statement is of the form

```
                  UNTIL   <comparison>
                              or
      UNTIL   <comparison>  AND/OR/XOR  <comparison>
```

The UNTIL statement is only valid when it follows a WHEN, WHENEVER, WAIT or EVERY statement. Its effect is to limit how long TIMELINER will patiently wait for fulfillment of the preceeding condition statement.

When the conditions in a condition statement which is followed by an UNTIL are failed, TIMELINER evaluates the conditions in the UNTIL statement. If these conditions are met, TIMELINER will skip over the action statements which immediately follow, and will start processing with the next condition statement.

An UNTIL statement with no <comparison> is accepted as legal input. When encountered, such an UNTIL statement is considered to have been satisfied, and TIMELINER will act accordingly.

Here is a sequence in which UNTIL statements are used to end a WHENEVER loop and to terminate a WAIT:

SEQUENCE FIVE

```
WHENEVER    ACC   >  32.0

UNTIL       MET   >  500.00

   <action statement>

   <action statement>

WAIT        1000

UNTIL       ROLLSTOP  =  ON

   <action statement>

   <action statement>
```

## 2.6 FOR Statement

The FOR line is of the form

```
FOR   <numeric>
```

where <numeric> is either a numeric literal or an integer/scalar variable. FOR is to WAIT and EVERY as UNTIL is to WHEN and WHENEVER. A FOR line is only valid when it follows a WHEN, WHENEVER, WAIT or EVERY statement.

When the condition statement which preceeds it is evaluated false, the FOR statement is also evaluated. If it is true, i.e. if the specified time has elapsed, the immediately following action statements are skipped and processing begins with the next condition statement.

Here is an example showing correct use of the FOR statement:

```
SEQUENCE SIX

    WHENEVER    ACC > 32.0

    FOR         100

        <action statement>

        <action statement>

    EVERY       10.0

    FOR         320.0

        <action statement>

        <action statement>
```

## 2.7 DO Statement

The DO statement consists of the single word "DO". It is required only for the case where condition statements are to be part of the overall action to be performed upon fulfillment of a WHENEVER or EVERY statement, or to be skipped upon fulfillment of an UNTIL or FOR statement.

When a group of condition and action statements are enclosed within a DO-END pair, those statements are all executed upon the fulfillment of a WHENEVER or an EVERY before return to the WHENEVER or EVERY line.

Similarly, if a condition statement has an accompanying UNTIL or FOR statement, then when the UNTIL or FOR is fulfilled before the preceeding condition statement, the DO-END enclosed statements will all be skipped over.

## 2.8 END Statement

The END statement consists of the single word "END". Its function is to terminate the grouping initiated by a DO statement.

Here is an example showing correct use of DO-END:

```
SEQUENCE SEVEN

   WHENEVER    ACC > 32.0

      DO

            <action statement>

            <action statement>

         WAIT         0.5

            <action statement>

            <action statement>

      END
```

## 2.9 CALL Statement

The CALL line is of the form

CALL <name of subseq>

where <name of subseq> is as described in Section 1.2. When encountered, the CALL statement will transfer control from the sequence where the CALL appears to the top of the subsequence. Upon completion of the subsequence control returns to the line after the CALL.

Note that when called a subseq behaves like a DO-END package. Thus it will all be executed upon fulfillment of an WHENEVER or an EVERY, and will all be skipped over upon fulfillment of an UNTIL or FOR.

A subseq may be called from any other seq or subseq. Because a subseq, like all TIMELINER input, is simply a script to be acted out, reentrancy is not a problem. A subseq can be called without worrying about whether the same subseq may still be in execution due to another call from another seq.

## 2.10 Comparison Expressions

Comparison expressions are components of WHEN, WHENEVER and UNTIL statements. They are used to specify conditions to be looked for. Comparison expressions may be of the form

<boolean/event variable><br>
or<br>
NOT <boolean/event variable>

However the more general form is

<noun> <comparer> <noun>

Where middle element <comparer> is one of the following:

= ¬= < >= < <=

Each side element <noun> may be either a boolean/event variable, an integer/scalar variable, a character/bit variable, a boolean/event literal, an integer/scalar literal, a character/bit literal, or a display page specification. These are described below.

### 2.10.1 Boolean/Event Variable

A boolean/event variable is any boolean or latched event which TIMELINER recognizes as a valid variable for decision making. A boolean/event variable is suitable for "=" or "¬=" comparison with a boolean/event literal or a boolean/event variable, and with a character/bit variable or a display page specification if it converts sucessfully to boolean form.

Note that specific arrangements must be made within the TIMELINER program to enable it to recognize a particular HAL variable by name.

Here are examples of comparison expressions involving boolean/event variables:

```
OMS_IGN_CMD

NOT OMS_IGN_CMD

OMS_IGN_CMD  ¬=  OMS_CUTOFF_CMD
```

### 2.10.2 Integer/Scalar Variable

An integer/scalar variable is any numeric variable which TIMELINER recognizes as a valid variable for decision making. An integer/scalar variable is suitable for any sort of comparison with a numeric literal, another integer/scalar variable, and when they convert sucessfully to a number, with a character/bit variable or a display page specification.

Besides its role in comparison expressions an integer/scalar variable may be the operand of a WAIT, EVERY or FOR statement.

Note that specific arrangements must be made within the TIMELINER program to enable it to recognize a particular HAL variable by name.

Here are examples of comparison expressions involving integer/scalar variables:

```
ALT  <  400000

MET  >=  T_CUTOFF
```

### 2.10.3 Character/Bit Variable

An character/bit variable is any character or (multiple) bit string which TIMELINER recognizes as a valid variable for decision making. A character/bit variable is suitable for "=" and "¬=" comparison with a character/bit literal, another character/bit variable, or a display page specification. When it converts sucessfully to a boolean, it may be compared to a boolean/event variable or a boolean/event literal. When it converts sucessfully to a numeric under the rules discussed in section 2.10.5, a character/bit variable may be compared in any way with a numeric literal, a integer/scalar variable, or (when they convert properly) another character/bit variable or a display page specification.

Note that specific arrangements must be made within the TIMELINER program to enable it to recognize a particular HAL variable by name.

Here are examples of comparison expressions involving character/bit variables:

```
JET_CONTROL_WORD  =  '0111011001010100'

   DISPLAY_TITLE  ¬=  'ORBIT COAST'

       SPEC_PAGE  =  '033'
```

### 2.10.4 Boolean/event Literal

A boolean/event literal is of the form "ON" or "OFF", or "TRUE" or "FALSE". The forms "'1'" and "'0'" are interpreted as character/bit literals, and are invalid as boolean/event literals. A boolean/event literal is suitable for "=" or "¬=" comparison with a boolean/event variable, and with a character/bit variable or a display page specification if it converts sucessfully to boolean form.

### 2.10.5 Numeric Literal

A numeric literal is any series of characters which give an integer or scalar number. A numeric literal is suitable for comparison with an integer/scalar variable, and, if they convert sucessfully to a numeric value, a character/bit variable or a display page specification.

Besides its role in comparison expressions a numeric literal may be the operand of a WAIT, EVERY or FOR statement.

Note that while in general a numeric literal must be a valid HAL "arithmetic literal" (see HAL Spec section 2.3.3), the days-hours-minutes-seconds form

DDD/HH:MM:SS.SS

and also the forms

HH:MM:SS.SS and MM:SS.SS

are specifically recognized as numeric literals and for use are simply converted to seconds.

### 2.10.6 Character/Bit Literal

A character/bit literal is any alphanumeric series at all enclosed by single quotes, and may contain blanks. A character/bit literal is suitable for "=" or "¬=" comparison with a character/bit variable, or a display page specification.

### 2.10.7 Display Page Specification

A display page specification is of the form

```
PAGE(L:C1-C2)
     or
CRT(L:C1-C2)
```

where L is a line number, C1 the beginning column number and C2 the concluding column number of a portion of the overall display page in the first case, or of the section of the display page devoted to the CRT.

Specific arrangements must be made with the TIMELINER program for it to recognize a variable by name. The PAGE/CRT capability is provided so that any of the information presented to the astronaut by the on-board displays can be used in a condition statement.

A display page specification is suitable for "=" or "¬=" comparison with a character/bit variable, a character/bit literal or another display page specification. When it converts sucessfully to a boolean, it may be compared to a boolean/event variable or a boolean/event literal. When it converts sucessfully to a numeric under the rules discussed in section 2.10.5, a display page specification may be compared in any way with a numeric literal, a integer/scalar variable, or (when they convert properly) a character/bit variable or another display page specification.

Here are examples of comparison expressions involving display page specifications:

```
CRT(3:47-50)    =    'EXEC'

CRT(1:38-50)   >=    001/12:5:33.3

CRT(1:38-50)    =    CRT(17:26-38)
```

## 3.0 ACTION STATEMENTS

The following action statements are currently implemented: ABORT, which terminates the simulation in progress; PRINT, used to print any recognized variable; LOAD, used to load (set) any recognized variable; KEY, used to input keystrokes; SEEK, used to seek further input to the TIMELINER program; and EXECUTE, used to invoke a recognized HAL procedure.

### 3.1 The ABORT Statement

The ABORT statement is of the form

ABORT

When executed this statement sets an event which immediately causes termination of the simulation.

### 3.2 The LOAD Statement

The LOAD statement is of the form

```
LOAD  <name of parameter>  <data>  <data>  ...  <data>
```

where <name of parameter> is a recognized HAL name, and <data> represents boolean, numeric or character/bit-string type data, for example three scalars if the parameter in question were a vector.

When executed this statement loads the data given, the amount of which must correspond to the quantity needed, into the specified parameter. Both the parameter's HAL name and its m/s id (if any) are printed at the time the LOAD takes place, along with the value to which it is loaded.

Unlike any other kind of statement, a LOAD statement may be continued on suceeding lines. If the data provided on the line with the word LOAD and the parameter specification is not sufficient to fill up the variable, a new line is read. Unless this line is obviously of some other type, in which case a complaint is issued, its contents are interpreted as data to be loaded into the parameter specified.

## 3.6 The EXECUTE Statement

The EXECUTE statement is of the form

EXECUTE <procedure>

where <procedure> is any HAL procedure which TIMELINER has been enabled to recognize.

The EXECUTE statement, when encountered, causes the specified HAL procedure to be invoked.

Note that the forms "'1'" and "'0'" are not valid for loading boolean variables. "ON" and "OFF" must be used instead.

Note that specific arrangements must be made within the TIMELINER program to enable it to recognize a particular HAL variable by name. A variable which TIMELINER recognizes as a valid LOAD variable is also recognized as a valid PRINT variable. However it does not automatically become a variable valid for decision-making.

## 3.3 The PRINT Statement

The PRINT statement is of the form

PRINT <name of variable or parameter>

where <name of variable or parameter> is the HAL name of any recognized HAL variable.

The PRINT statement causes the variable specified to be printed. Any variable that TIMELINER has been enabled to recognize for purposes of LOADing, is also available for printing.

## 3.4 The KEY Statement

The KEY statement is of the form

KEY <keystroke> <keystroke> ... <keystroke>

where <keystoke> is any key that exists on the Shuttle keyboard. See section 5 of the Level A FSSR for a description of the crew-interface keystroke language. The following figure (Level A FSSR figure 3-50) illustrates the keyboard itself.

# User's Guide to the TIMELINER Language

(Simulation Version)

Revision 3.0

Don Eyles

Concetta Cuevas

March 27, 1992

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

## 1.0 INTRODUCTION

This document describes a language called TIMELINER that provides capabilities allowing a user to initialize, control and debug a simulation by means of a script written ahead of time.

This document serves both as a User's Guide and as the principal descriptive document for the version of the Ada-language TIMELINER language designed for use in ground-based simulations.

The TIMELINER language is under consideration by NASA for use as the onboard "user interface language" (UIL) for the Space Station Freedom. Features designed expressly for the SSF version are not described in this document. However, the script organization and control statements are identical in all versions of the TIMELINER language.

Users constitute the best judges of any system such as TIMELINER that is designed to be "easy to use". User comments are therefore solicited, both with respect to the language and to this User's Guide. We will do our best to respond to comments with appropriate changes.

### 1.1 Purpose of TIMELINER

TIMELINER was created because of the inadequacy of the software tools that were available 10 years ago at the Draper Laboratory for supplying initialization information to simulations of space systems.

TIMELINER provides the capability both for time-zero initialization and for the input of information during the simulation at times or under conditions that are specified by the user. TIMELINER scripts can be created to insert perturbations during a run or to play the role of a "paper pilot".

In addition to its function as a flexible initialization tool, the TIMELINER language can be used as a debugging aid. For example, a TIMELINER sequence may be constructed that will assist the capture of information needed to analyze an anomaly.

### 1.2 Basic Principles

The central concept of the TIMELINER language is that the user must be allowed to specify an arbitrary number of independent sequences. Within each sequence processing proceeds sequentially from top to bottom as defined by the constructs used. Meanwhile the various sequences operate in parallel with each other.

Such a "serial/parallel" capability allows input streams to be organized in accordance with whatever principles seem appropriate to the user. The user is freed from the necessity to integrate all functions into a single input stream.

TIMELINER streams may be broken down by discipline (e.g. flight control, guidance, crew input) or by any other rule that is convenient.

TIMELINER includes the capability of creating "subsequences" that may be called by another TIMELINER sequence.

A particular timeline sequence may be quite small: for example a single "whenever" construct specifying an action to occur whenever a particular condition occurs. On the other hand a sequence may be quite long: for example a long string of "when" constructs specifying actions to be taken, in turn, as a series of conditions become true.

## 1.3 Notation

This section describes the notation that is used in this document to describe the TIMELINER language:

(a) Reserved words including statement keywords are denoted by words printed in upper-case letters, for example: SEQUENCE, WHEN, AS.

(b) Syntactical elements including statement "components" are denoted by the use of a phrase in lower-case letters enclosed by corner brackets, for example: <name_of_sequence>, <singular_numeric>, <boolean_combo>. Such syntactical elements are described in Section 7.0.

(c) Square brackets are used to enclose optional elements. The elements within brackets may occur once at most.

(d) Braces (curly brackets) are used to enclose repeated elements. The elements within braces may appear zero or more times.

(e) A vertical bar ("|") separates alternative elements.

Note that TIMELINER is case insensitive. That is, the compiler will accept upper-case, lower-case, or mixed forms of every reserved word; every action or parameter name; every function or constant name; and every name created by a statement. The listing created at compile time, and all displays created during execution time, will use upper-case letters.

## 1.4 How to Use TIMELINER

TIMELINER operation has two parts:

*Compile time:*

At "compile time" raw TIMELINER scripts provided by the user are parsed, checked for errors, and tabulated. A compile-time "listing" is provided. If free from errors the script is stored in a file in a form suitable for execution. The name of the executable file is determined by the name of the "bundle" that constitutes the uppermost level of the TIMELINER hierarchy.

At compile time TIMELINER operates in "batch" fashion. At the Draper Laboratory the TIMELINER compile capability is installed on the MicroVAX computer that serves as the "development host" for the Draper Lab real-time test bed of the Space Station DMS. Compile capability may also be hosted by other platforms, including individual Macintosh computers.

*Run time:*

At "run time" the executable data corresponding to a particular TIMELINER "bundle" is read from its file and executed. During execution-time TIMELINER is called repetitively, at frequent intervals. The spacing of these intervals determines the "granularity" with which TIMELINER is capable of timing functions such as WAIT.

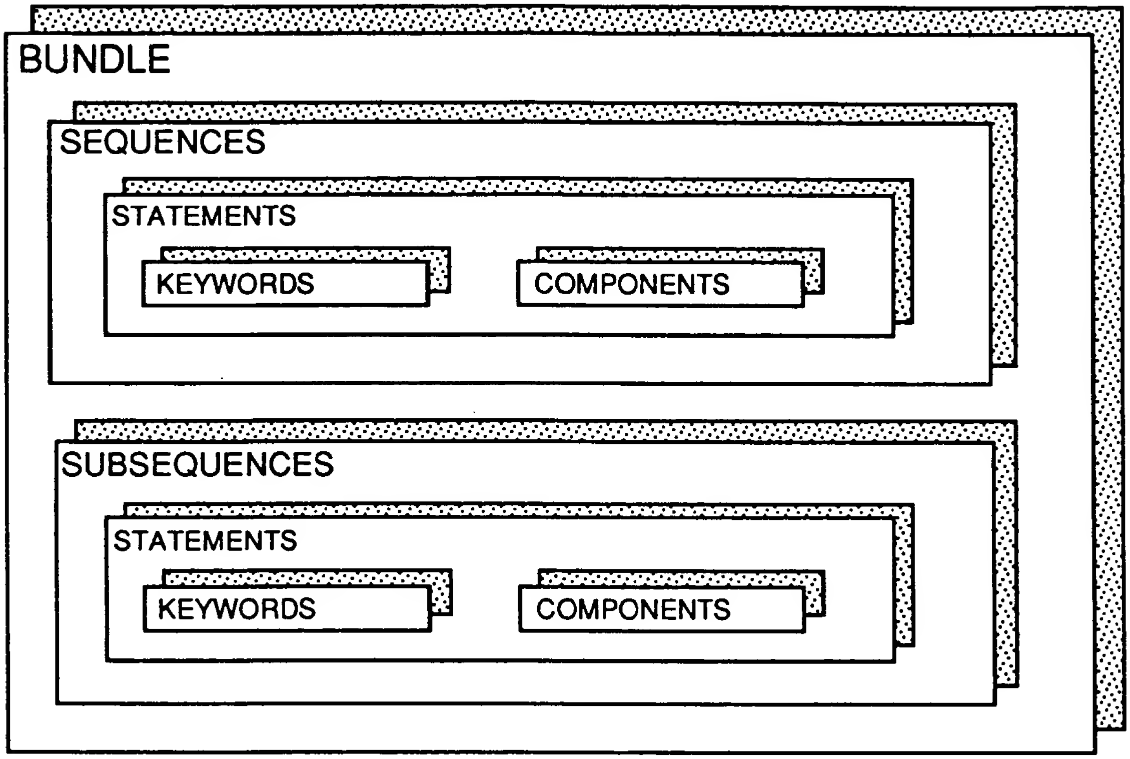
TIMELINER prints TIMELINER statements when they are executed. This execution-time printing forms part of the text output stream of the simulation that can be displayed on a terminal or printed on paper.

## 2.0 STRUCTURE OF TIMELINER SCRIPTS

This section describes the hierarchical structure of a TIMELINER script. The "blocking statements" that implement this structure are described in the following chapter.

The highest level of the TIMELINER hierarchy is the "bundle" -- used for grouping and packaging the "sequences" and "subsequences" that constitute the substance of a TIMELINER script. Sequences and subsequences, in turn, are made of "statements".

The TIMELINER hierarchy is illustrated by the following picture:



### 2.1 The "Bundle"

The uppermost hierarchical level of a TIMELINER script is the "bundle". The concept of "bundle" is new to the Ada-language version of TIMELINER.

The "bundle" is defined as a grouping of sequences and subsequences that form a whole, either because of some common purpose, or because they are targeted for use in a particular operating environment.

A bundle may contain any number of sequences and subsequences. However, a subsequence must lie within the same bundle as the CALL statement that invokes it.

A TIMELINER script may contain any number of bundles. A separate executable-code file is produced for each bundle. The name of the file created for each bundle is

TL_<name>.DATA

where <name> is an alphanumeric word without imbedded blanks chosen by the user. Underscores are allowed. In the present implementation such names are limited to 40 characters.

A bundle is initiated by a BUNDLE header statement (described in Section 3.1 below) and terminated by a CLOSE BUNDLE statement (Section 3.4).

Note that all TIMELINER statements, except comments and "directives", must lie inside a bundle. Therefore the first statement in any TIMELINER script is normally a BUNDLE header statement and the last is normally a CLOSE BUNDLE statement.

In some cases the user is not interested in dividing TIMELINER processing into bundles. In this case no bundle header or closer is required. The compiler will place the executable data in a file named TL_SCRIPT.DATA as though a "bundle" had been defined with the name SCRIPT.

(In the space station version of TIMELINER "bundles" may dynamically be brought into memory for execution ("installed") or deleted from memory ("removed"). A delta-time and Ada-priority may be attached to a bundle when it is installed. However, these capabilities are not available in the current simulation version of TIMELINER.)

## 2.2 The "Sequence"

A given TIMELINER bundle contains one or more "sequences" and "subsequences". Sequences and subsequences in turn contain TIMELINER statements.

The essence of a sequence is to establish an independent stream of execution. That is, a sequence is processed in parallel with other sequences. The TIMELINER user may organize a script into any number of sequences as required to accomplish his purpose.

Parallelism among TIMELINER sequences offers several advantages. When two or more conditions requiring a response may occur in an order that is unknown ahead of time -- a situation that would be very difficult to handle within a single sequential stream -- a TIMELINER user may set up multiple sequences to deal with the disparate situations. A new function does not have to be integrated into an existing stream of execution. It may be implemented as an additional sequence.

A TIMELINER script may be as short as a single WHENEVER construct that performs an action whenever a specified condition occurs, or as long as a string of WHEN constructs and WAIT statements that provides the master sequencing for the simulation of an entire mission. Many TIMELINER scripts tend to be organized into sequences of these two types.

A sequence is initiated by a SEQUENCE header statement (described in Section 3.2 below) and terminated by a CLOSE SEQUENCE statement (Section 3.4).

Note that, aside from comments, all TIMELINER statements except those that initiate and conclude a bundle must lie within some sequence or subsequence. Therefore the first statement in any TIMELINER bundle, other than comments and directives, must be a sequence (or subsequence) header statement.

## 2.3 The "Subsequence"

A TIMELINER "subsequence" is in many ways similar to a sequence. The difference is that while a sequence always forms a new stream of execution, a subsequence is executed within the stream of execution created by the sequence that calls it by means of a CALL statement (Section 4.11).

Subsequence calls may be nested. That is, a subsequence may call another subsequence. In every case, however, a subsequence forms part of the stream of execution established by the sequence that makes the upper-level call. A subsequence that is never called will compile -- but of course it cannot be executed.

Note that while subsequence calls may be nested, subsequences themselves appear in a TIMELINER script at the same level as sequences. A subsequence should not be placed physically inside a sequence.

A subsequence must belong to the same bundle as the sequence(s) or subsequence(s) that call it. That is, subsequence calls may not be made across bundle boundaries. An error message (Cuss 41) is issued at TIMELINER compile-time, during processing of the CLOSE BUNDLE statement, if a called subsequence is not found in the bundle.

The statements making up the subsequence have the same effect as they would have if they physically were part of the calling sequence. However, constructs must be fully contained within a subsequence. That is, a WHEN construct (for example) may not be initiated in a sequence and closed in a called subsequence, nor vice versa.

Users typically employ subsequences to encapsulate particular bundles that may be required as part of a sequence that performs upper-level (e.g. mission) sequencing. This simplifies the appearance of the upper-level sequence and removes the particular bundle, which may seldom change, from the upper-level sequence where changes may be more frequent.

A subsequence is initiated by a SUBSEQUENCE header statement (described in Section 3.3 below) and terminated by a CLOSE SUBSEQUENCE statement (Section 3.4).

## 2.4 "Statements"

Bundles contain sequences and subsequences. Sequences and subsequences, in turn, contain statements.

A TIMELINER statement must start on a new line in the raw script, and may continue onto additional lines. In the present implementation a line is limited to 132 characters and the total length of a statement is limited to 660 characters.

A statement type is recognized by means of the keyword that begins the statement. This keyword must be the first word on a line for it to be recognized. For example the WHEN statement is recognized when a line is encountered whose first word is "WHEN".

TIMELINER statements are of four types: "blocking" statements, "control" statements, "action" statements, and "non-executable" statements.

Blocking statements are those used to initiate and conclude TIMELINER bundles, sequences, and subsequences. These statements are described in Chapter 3 of this User's Guide.

Control statements are those associated with the WHEN, WHENEVER, EVERY and IF constructs that are used to specify the conditions for TIMELINER actions. The WAIT and CALL functions are also considered to be "control" statements. Control statements are described in Chapter 4.

Action statements stand alone in themselves and perform some "action" such as loading or printing a variable. LOAD and PRINT, for example, are action statements. Statements such as START, STOP, and RESUME, which are used to activate and deactivate sequences, are also classed as action statements. Action statements are described in Chapter 5.

Non-executable statements are those statements that declare or define a name for internal use. Non-executable statements are described in Chapter 6.

Note: In addition to the TIMELINER statements summarized above, the TIMELINER compiler accepts "compiler directives". Compiler directives are described below in Section 2.7.

## 2.5 "Components"

Once the statement type is known, the compiler evaluates the remainder of the statement. Every statement consists of components and reserved words. TIMELINER reserved words are described in Section 7.1 of this User's Guide.

"Component" is a general term used to cover a variety of "things" that may occur in a TIMELINER statement.

Most components are of one of three types: boolean, numeric, and character string.

Within each type, a component may be a literal, a list, a combination, a definition, a combination, a variable defined by the target system (i.e. the simulation), a declared internal variable, a built-in function, or a built-in constant. Several special-purpose components also exist, such as the "wild card", an asterisk that may be used in a subscript expression.

A third characteristic of components, that cuts across the two dimensions just described, is the size of the component. The size of a component is its total number of elements. Component size is one-dimensional. Therefore, for example, TIMELINER does not object if a 9-element array is assigned to a 3x3 array.

TIMELINER also recognizes "range" components. A range component establishes a numeric range that can be used as a subscript or compared to a numeric component in a control statement such as IF.

Component types are described more fully in Section 7.4 of this document.

## 2.6 Comments and Blank Lines

TIMELINER permits comments to be added to a script. Such comments are visible in the raw script and in the compile-time listing of the script. As in any computer language, comments are not retained in the executable code.

The Ada-language version of TIMELINER adopts the Ada language's convention in regard to comment designation. Two or more adjacent hyphens (i.e. minus signs) mark the beginning of a comment. All material between the first double hyphen on a line and the end of the line is considered to be part of the comment. A comment may occupy a line by itself, or share a line with a TIMELINER statement.

Blank lines are legal and the spacing implied by such blank lines is retained in the compile-time listing of the script.

## 2.7 Compiler Directives

The TIMELINER compiler allows the user to control certain characteristics of the compiler. All compiler directives begin with the keyword "DIRECT", which may also be written as "DIRECTIVE".

A compiler directive may appear at any point in the raw input script. It will take effect when it is encountered.

The following compiler directives are supported by the present implementation of TIMELINER:

*Print Level*

The printing that occurs as part of a TIMELINER compilation is controlled by a compiler directive of the form:

```
      DIRECT[IVE]  PRINT_LEVEL  n
```

where "n" is a zero or positive integer. The higher the integer the more printing occurs. If "n" is zero no optional printing occurs. If "n" is 10 maximum printing occurs. The printing evoked by this directive may not be cleanly formatted. Much of this printing is of value only for debugging the compiler, and therefore of little interest to users.

Note that multiple print level directives may be used, and therefore printing may be turned on selectively for some portion of the script.

Setting print level to 1 at any time before the end of the script causes a formatted summary of the data created by the compilation to be output at the end of the compilation listing of the script.

The default print level is zero.

*Script name*

The file produced by the TIMELINER compilation is normally given a name based on the name in the BUNDLE header statement. This name can be overridden by means of the following directive:

```
DIRECT[IVE]  SCRIPT_NAME  <name>
```

If this directive is present the file produced by the compilation will be given the name "TL_<name>.DATA".

*Optimization*

The TIMELINER compiler contains logic that attempts to reduce the memory required to store a script. Optimization is invoked by means of the following directive:

```
DIRECT[IVE]  OPTIMIZE
```

Optimization is turned off by the following directive:

```
DIRECT[IVE]  NO_OPTIMIZE
```

The optimization strategy is to identify components that have already been filed and to reuse these components whenever possible. When optimization is turned on compilation processing will take longer, but the resulting file will be smaller.

Note that at the present time the optimization capability of the TIMELINER compiler is extremely rudimentary. Further optimization capability will be added at a future time.

The default optimization status is OFF.

*Check Variable List*

TIMELINER is given knowledge of simulation variables by means of a "variable list" that contains the names, the addresses, and other data regarding each variable that must be referred to in a TIMELINER script. (See section 9.2 for more information about the variable list required by TIMELINER.)

The variable list must be maintained manually, and therefore is subject to errors. The most common error is in the alphabetization of the variables.

If the raw script contains the directive:

```
DIRECT[IVE]  CHECK_VAR_LIST
```

then the variable list will be checked for errors. The variable list checking capability can detect the following errors:

(1) mis-alphabetization,

(2) dimension error, i.e. an error in which the number of dimensions for a variable is set to a value less than zero or greater than the allowed maximum (currently 3),

It is recommended that following every modification of the variable list, this directive be added to a script to verify that the modification did not introduce one of these errors.

## 3.0 BLOCKING STATEMENTS

This chapter describes the "blocking" statements that are used to initiate and conclude TIMELINER bundles, sequences, and subsequences.

### 3.1 The BUNDLE Header Statement

The uppermost hierarchical level of the Ada-language version of TIMELINER is the "bundle". The role of the "bundle" is discussed in the previous chapter.

The start of a bundle is marked by a header statement of the form:

```
BUNDLE <name_of_bundle>
```

where <name_of_bundle> is an alphanumeric word without imbedded blanks chosen by the user. Underscores are allowed. In the present implementation such names are limited to 40 characters.

A bundle header statement is never "executed", as such. However, the bundle name determines the name of the file to which the compile-time processing outputs the executable form of the script. The name of the file created for each bundle is

```
TL_<name_of_bundle>.DATA
```

If a raw script contains multiple bundles then multiple executable files are created.

Each bundle is closed by a CLOSE BUNDLE statement, as described below in Section 3.4.

### 3.2 The SEQUENCE Header Statement

Multiple sequences, which execute in parallel with each other, contain the substance of a TIMELINER script. The role of the "sequence" is discussed in the previous chapter.

The start of a sequence is marked by a header statement of the form:

```
SEQ[UENCE]  <name_of_sequence>  [ACTIVE | INACTIVE]
```

where <name_of_sequence> is an alphanumeric word without imbedded blanks chosen by the user. In the present implementation such names are limited to 40 characters.

The word ACTIVE or INACTIVE may be added at the end of the sequence header statement. This word determines whether the sequence is in an active or an inactive state at the start of the run. The default is ACTIVE. If the sequence starts out INACTIVE, it will not run unless it is activated by means of the START statement (Section 5.6).

Each sequence is closed by a CLOSE SEQUENCE statement, as described below in Section 3.4.

## 3.3 The SUBSEQUENCE Header Statement

Subsequences are like sequences except that they form part of the stream of execution of the sequence that calls the subsequence. The role of the "subsequence" is discussed in the previous chapter.

The start of a subsequence is marked by a header statement of the form:

```
      SUBSEQ[UENCE]  <name_of_subsequence>
```

where <name_of_subsequence> is an alphanumeric word without imbedded blanks chosen by the user. In the present implementation such names are limited to 40 characters.

Each subsequence is closed by a CLOSE SUBSEQUENCE statement, as described below in Section 3.4.

## 3.4 The CLOSE Statement

The CLOSE statement is used to conclude TIMELINER bundles, sequences, and subsequences.

The end of a bundle is marked by the statement

```
      CLOSE  BUNDLE  [<name_of_bundle>]
```

When compile-time processing reaches a CLOSE BUNDLE statement the file containing the executable code for the bundle is closed. A new file will be opened if the script contains any additional bundles. This statement has no function during execution-time.

The end of a sequence is marked by the statement

```
      CLOSE  SEQ[UENCE]  [<name_of_sequence>]
```

When execution-time processing reaches a CLOSE SEQUENCE statement the sequence in question is concluded and placed in an inactive state.

The end of a subsequence is marked by the statement

```
      CLOSE  SUBSEQ[UENCE]  [<name_of_subsequence>]
```

When execution-time processing reaches a CLOSE SUBSEQUENCE statement processing returns to the sequence or subsequence containing the CALL statement that invoked the subsequence, at the statement immediately following the CALL statement.

In all three types of the CLOSE statement, the user may optionally include the name of the block being closed as the third word in the statement. In some cases providing this information may make a script easier to read. If the block name is included, TIMELINER will issue an error message (Cuss 28) if the name that is given does not match the name of the innermost open block.

### 3.5 Example of TIMELINER Blocking

The following example illustrates the upper level organization of a TIMELINER script:

```
BUNDLE MISSION_SEQUENCING

      SEQUENCE MAIN
            <statement>
            CALL LOADER
            <statement>
      CLOSE SEQUENCE MAIN

      SEQUENCE SECONDARY
            <statement>
            <statement>
      CLOSE SEQUENCE SECONDARY

      SUBSEQUENCE LOADER
            <statement>
            <statement>
      CLOSE SUBSEQUENCE LOADER

CLOSE BUNDLE MISSION_SEQUENCING
```

The executable version of this script would be placed in a file of the name:

```
TL_MISSION_SEQUENCING.DATA
```

## 4.0 CONTROL STATEMENTS

This chapter describes the "control" statements provided to permit the TIMELINER user to specify the conditions under which actions are to occur.

Some TIMELINER control statements initiate "constructs" -- namely the WHEN, WHENEVER, EVERY and IF statements. Such constructs are concluded by an END statement. Other control statements, such as BEFORE, WITHIN, OTHERWISE, ELSE, and ELSE IF, are used within these constructs. Two additional, stand-alone statements are classified as control statements, namely WAIT and CALL.

## 4.1 The WHEN Construct

The WHEN statement is used to specify conditions at which an action is to occur -- on a one-shot basis. The WHEN statement initiates a WHEN construct.

All forms of the WHEN construct contain a "condition" expressed as a <singular_boolean> component -- that is, an unarrayed component that can be evaluated as TRUE or FALSE. TIMELINER components are discussed below in Section 7.4.

### 4.1.1 The One-Line WHEN Construct

The simplest form of the WHEN construct consists of a single statement of the form:

```
WHEN <singular_boolean> [THEN] CONTINUE
```

This statement, when encountered in a given sequence during execution, causes the sequence to pause until the condition expressed as a <singular_boolean> component is true. When the condition is met ✓ execution continues with the next statement.

### 4.1.2 The Simple WHEN Construct

The WHEN construct may also be written as follows:

```
WHEN  <singular_boolean>
      <statements>
END  [WHEN]
```

The WHEN statement, when encountered, causes the sequence of which it is a part to pause until the condition expressed as a <singular_boolean> component is true. When the condition is met, the statements following the WHEN statement are processed until the END statement is encountered. Processing then "drops through" to the statements following the END statement.

The statements within a WHEN construct may be any action or control statement, including nested control constructs.

WHEN statements are most useful in the case of a TIMELINER sequence being used to specify a long string of actions. Each WHEN construct may be thought of as a gate across a straight road. The stated condition must occur in order for the gate to allow the traveler to proceed.

Here is an example of the use of the simple WHEN construct:

```
      WHEN ACCEL < 0.1
            <statement>
            <statement>
      END WHEN
      WHEN TIME >= 11:11:11 OR ABORT_FLAG = TRUE
            <statement>
            <statement>
      END WHEN
```

In this case the sequence waits until the acceleration denoted by the variable ACCEL is less than 0.1, then executes the statements within the construct. The sequence then waits until either time reaches 11 hours, 11 minutes, and 11 seconds, or the boolean ABORT_FLAG becomes true; and then executes the statements within the second WHEN construct. The sequence then continues at the statements that follow.

### 4.1.3 The Modified WHEN Construct

Additional forms of the WHEN construct may be created by use of the BEFORE or WITHIN statement. Furthermore, if WHEN is modified by BEFORE or WITHIN, an OTHERWISE statement may be used.

The modified form of the WHEN construct is as follows:

```
      WHEN  <singular_boolean>
         [BEFORE  <singular_boolean>  |  WITHIN  <singular_numeric>]
            <statements>
         [OTHERWISE
            <statements>]
      END  [WHEN]
```

The WHEN statement, when encountered, causes the sequence of which it is a part to pause until the condition expressed as a <singular_boolean> is true. When the condition is met, the statements following the BEFORE or WITHIN statement are processed until either an OTHERWISE or the END statement is encountered. Processing then continues at the statements following the END statement.

However, each time the WHEN condition is to be checked, the condition specified by the BEFORE or WITHIN statement is evaluated. Regardless of whether the WHEN condition is fulfilled simultaneously, if the condition that forms part of the BEFORE statement is fulfilled, or if the time interval specified in the WITHIN statement has elapsed since the WHEN statement was encountered, then processing continues at the statements following the OTHERWISE statement. If there is no OTHERWISE statement, processing skips straight to the END statement and continues at the statements following the END.

In other words, the BEFORE and WITHIN statement may be used to terminate operation of the WHEN construct if a condition (BEFORE) is met, or if a time interval (WITHIN) elapses. An OTHERWISE statement allows a separate set of statements to be provided that will be processed when the construct is terminated.

A modified WHEN construct may also be thought of as executing separate statements depending upon which of two conditions occurs first, such as in the English statement:

> When the pot boils before the timer buzzes, turn off the heat, otherwise reset the timer.

or the statement:

> When the pot boils within 10 minutes, turn off the heat, otherwise turn up the heat.

The statements within the WHEN construct, including those in the OTHERWISE clause, may be any action or control statements, including nested control constructs.

To return to the metaphor of the WHEN statement as a gate across a straight road, fulfillment of a BEFORE or WITHIN condition creates a path that allows the traveler to bypass the obstacle if the specified condition is met. The OTHERWISE clause, if present, lies along the bypass path.

Here is an example of the modified WHEN construct:

```
WHEN ACCEL < 0.3
   WITHIN 10
      <statement>
      <statement>
END WHEN
```

In this example the sequence waits until the acceleration given by the variable ACCEL is less than 0.3, or until 10 seconds have elapsed since that wait began. If the time interval elapses before or at the same time as the WHEN condition is met, then processing skips to the END statement (and to the statements that follow) without executing the inside statements. If the ACCEL condition is met first, the inside statements are executed before reaching the END statement.

Here is another example of the modified WHEN construct, this time including an OTHERWISE clause:

```
WHEN VIEW_ANGLE > 15
   BEFORE LENS_TEMP > 212 or SENSOR_VOLTS > 25.4
      <statement>
      <statement>
   OTHERWISE
      <statement>
      <statement>
END WHEN
```

In this example two conditions are waited for, the WHEN condition expressed in terms of VIEW_ANGLE, and the BEFORE condition involving lens temperature and sensor voltage. If the WHEN condition occurs first the first set of inside statements is executed. Otherwise the second set is executed. In either case processing then proceeds to the END statement and to the statements that follow it.

## 4.2 The WHENEVER Construct

The WHENEVER statement is used to specify conditions at which an action is to occur -- on a repeating basis. The WHENEVER statement initiates a WHENEVER construct.

All forms of the WHENEVER construct contain a "condition" expressed as a <singular_boolean> component -- that is, an unarrayed component that can be evaluated as TRUE or FALSE. TIMELINER components are discussed below in Section 7.4.

A WHENEVER construct in its simplest form consists of a WHENEVER statement, an END statement, and the statements that lie between. The simple WHENEVER construct is described below in Section 4.2.1.

A WHENEVER construct may optionally be modified by a BEFORE or WITHIN statement. The modified WHENEVER construct is described below in Section 4.2.2.

### 4.2.1 The Simple WHENEVER Construct

The simple form of the WHENEVER construct is as follows:

```
WHENEVER  <singular_boolean>
     <statements>
END  [WHENEVER]
```

The WHENEVER statement, when encountered, causes the sequence of which it is a part to pause until the condition expressed as a <singular_boolean> component is true. When the condition is met, the statements following the WHENEVER statement are processed until the END statement is encountered. Then processing returns to the WHENEVER statement. Upon the next off-to-on transition of the condition stated in the WHENEVER statement the interior statements are again processed. This loop repeats indefinitely.

The statements within a WHENEVER construct may be any action or control statements, including nested control constructs.

WHENEVER constructs are most useful when a TIMELINER sequence is being used to perform certain actions whenever a particular condition is fulfilled. A WHENEVER construct may be thought of as a gate across a circular road. The stated condition must occur in order for the gate to open, but the traveler will encounter the same gate again and must again wait for the condition to become true.

Here is an example of the use of the simple WHENEVER construct:

```
WHENEVER JET_FAIL = TRUE
      <statement>
      <statement>
END WHENEVER
```

In this example, when the sequence reaches the WHENEVER construct a loop is started that will process two statements at any time that a jet failure occurs. If, following processing of the statements inside the construct, the jet failure indication is still present, the statements will not be processed again immediately. They will be processed again if the jet failure is reset and subsequently reappears.

(Note that if the flag is turned off and on during the time that the interior statements are being processed, these transitions will not be sensed by the WHENEVER construct. The construct will act as though the condition were continuously present.)

### 4.2.2 The Modified WHENEVER Construct

More complicated forms of the WHENEVER construct may be created by use of the BEFORE or WITHIN statement..

The modified form of the WHENEVER construct is as follows:

```
WHENEVER  <singular_boolean>
   [BEFORE  <singular_boolean>  |  WITHIN  <singular_numeric>]
      <statements>
END  [WHENEVER]
```

The WHENEVER statement, when encountered, causes the sequence of which it is a part to pause until the condition expressed as a <singular_boolean> component is true. When the condition is met, the statements following the BEFORE or WITHIN statement are processed until the END statement is encountered. Then processing returns to the WHENEVER statement. The loop repeats upon the next off-to-on transition of the WHENEVER condition.

However, each time the WHENEVER condition is to be checked, the condition established by the BEFORE or WITHIN statement is evaluated. Regardless of whether the WHENEVER condition is fulfilled, if the condition that forms part of the BEFORE statement is fulfilled, or if the time interval specified in the WITHIN statement has elapsed since the WHENEVER statement was first encountered, then processing skips straight to the END statement and drops through to the statements that follow the END statement.

In other words, the BEFORE and WITHIN statement may be used to terminate the loop created by the WHENEVER statement if a (BEFORE) condition is met, or if a (WITHIN) time interval elapses. Without a BEFORE or WITHIN statement a WHENEVER construct will loop indefinitely.

The modified WHENEVER construct is useful when a TIMELINER sequence needs to perform certain actions whenever a condition is fulfilled, up until some other condition occurs or until a time interval elapses. Returning to the analogy of a gate across a circular road, a BEFORE or WITHIN clause establishes a path out of the circle that will be taken upon fulfillment of the BEFORE or WITHIN condition.

Here is an example of the modified WHENEVER construct:

```
WHENEVER DAP_MODE = "AUTO"
   BEFORE MAJOR_MODE /= 301
      <statement>
      <statement>
END WHENEVER
WHENEVER DAP_MODE = "AUTO"
   BEFORE MAJOR_MODE /= 302
      <statement>
      <statement>
END WHENEVER
```

In this case the sequence processes two statements whenever digital autopilot mode (a character string) goes to automatic, for as long as MAJOR_MODE is 301. Execution then moves to the second WHENEVER construct, which executes different statements whenever autopilot mode goes to automatic during major mode 302.

## 4.3 The EVERY Construct

The EVERY statement is used to make some actions occur repetitively at some specified time interval. The EVERY statement initiates an EVERY construct.

All forms of the EVERY construct contain a time interval expressed as a <singular_numeric> component -- that is, an unarrayed component that can be evaluated as a number. TIMELINER components are discussed below in Section 7.4.

An EVERY construct in its simplest form consists of an EVERY statement, an END statement, and the statements that lie between. The simple EVERY construct is described below in Section 4.3.1.

An EVERY construct may optionally be modified by a BEFORE or WITHIN statement. The modified EVERY construct is described below in Section 4.3.2.

### 4.3.1 The Simple EVERY Construct

The simple form of the EVERY construct is as follows:

```
EVERY  <singular_numeric>
      <statements>
END  [EVERY]
```

The EVERY statement, when first encountered, immediately drops through to execute the statements that lie between the EVERY statement and the

END statement. When the END statement is reached, processing returns to the EVERY statement. When the time interval expressed as a <singular_numeric> component has elapsed, since the previous encounter with the EVERY statement, the interior statements are again processed. This loop repeats indefinitely.

The time interval in an EVERY statement may be specified as a literal or a variable. If the time interval is specified as a variable, the variable is re-evaluated each time the EVERY statement is processed.

The statements within an EVERY construct may be any action or control statements, including nested control constructs.

An EVERY constructs is most useful when a TIMELINER sequence is being used to perform certain actions at intervals. An EVERY construct may be thought of as a gate across a circular road which opens at intervals.

Here is an example of the use of the simple WHENEVER construct:

```
EVERY DT_NAV
      <statement>
      <statement>
END EVERY
```

In this example, when the sequence reaches the EVERY construct a loop is started that will process two statements at intervals corresponding to the navigation delta-time given by the variable DT_NAV. The loop continues indefinitely.

### 4.3.2 The Modified EVERY Construct

More complicated forms of the EVERY construct may be created by use of the BEFORE or WITHIN statement.

The modified form of the EVERY construct is as follows:

```
EVERY  <singular_numeric>
   [BEFORE  <singular_boolean>   |   WITHIN  <singular_numeric>]
      <statements>
END  [EVERY]
```

The EVERY statement, when first encountered, immediately drops through to execute the statements that lie between the BEFORE or WITHIN statement and the END statement. When the END statement is reached, processing returns to the EVERY statement. When the time interval specified by the EVERY statement has elapsed, the interior statements are again processed.

However, on each pass the condition established by the BEFORE or WITHIN statement is evaluated. If the condition that forms part of the BEFORE statement is fulfilled, or if the time interval specified in the WITHIN statement has elapsed since the EVERY statement was first encountered, then processing skips straight to the END statement and drops through to the statements that follow the END statement.

In other words, the BEFORE and WITHIN statement may be used to terminate the loop created by the EVERY statement if a (BEFORE) condition is met, or if a (WITHIN) time interval elapses. Without a BEFORE or WITHIN statement an EVERY construct will loop indefinitely.

The modified EVERY construct is useful when a TIMELINER sequence needs to perform certain actions at intervals for some period of time or until some condition occurs. Returning to the analogy of a gate across a circular road, a BEFORE or WITHIN clause establishes a path out of the circle that will be taken upon fulfillment of the BEFORE or WITHIN.

Here is an example of the modified EVERY construct:

```
EVERY 1
   WITHIN 300
      <statement>
      <statement>
END EVERY
EVERY 2
   BEFORE MAJOR_MODE /= 302
      <statement>
      <statement>
END EVERY
```

In this case the sequence processes two statements every one second for 5 minutes. Execution then moves to the second EVERY construct, which executes different statements every 2 seconds until major mode is no longer 302.

The user may find it useful to nest an EVERY construct inside other constructs such as WHEN or WHENEVER. Consider the example:

```
WHENEVER JET_FAIL = ON
      EVERY 0.05
         WITHIN 0.5
            PRINT AVAILABLE_JETS
      END EVERY
END WHENEVER
```

Whenever a jet failure occurs the nested EVERY construct will print the jet availability matrix every 50 ms. for half a second. Eleven printings will occur.

## 4.4 The IF Construct

The IF statement is used to choose among actions -- at a particular point in time. The IF statement initiates an IF construct.

Unlike the WHEN, WHENEVER, and WAIT constructs, an IF construct is not tied in any way to the passage of time. The choice embodied in the IF statement is processed all at once.

All forms of the IF construct contain a "condition" expressed as a <singular_boolean> component -- that is, an unarrayed component that can be evaluated as TRUE or FALSE. TIMELINER components are discussed below in Section 7.4.

An IF construct in its simplest form consists of an IF statement, an END statement, and the statements that lie between. An IF construct may optionally contain multiple ELSE IF clauses and an ELSE clause.

The form of the IF construct is as follows:

```
IF  <singular_boolean>
     <statements>
{ELSE  IF  <singular_boolean>
     <statements>}
[ELSE
     <statements>]
END  [IF]
```

When the IF statement is encountered, its condition expressed as a <singular_boolean> component is evaluated. If the condition passes, the statements that follow the IF statement are processed, and execution then skips to the END statement and falls through to whatever statements follow. If the condition fails, and there is no ELSE IF or ELSE clause, execution skips straight to the END statement.

If there are any ELSE IF or ELSE clauses, they are processed in the obvious way. That is, if the IF condition fails, execution skips to the first ELSE IF or ELSE. If it is an simple ELSE clause the statements lying between the ELSE and the END statement are processed. If it is an ELSE IF clause, the specified condition is evaluated and if true the statements lying between that ELSE IF and the next ELSE IF or ELSE or END are processed. This process continues until the IF construct is completed.

The statements within an IF construct may be any action or control statements, including nested control constructs.

Here is an example of the use of the simple IF construct:

```
IF INERTIAL_POSITION(3) > 0
     <statement>
     <statement>
END IF
```

In this example, certain statements are executed if and only if the spacecraft is currently over the northern hemisphere.

Here is a more elaborate example of an IF construct:

```
IF MAJOR_MODE = 301
     <statement>
ELSE IF MAJOR_MODE = 302
     <statement>
ELSE IF MAJOR_MODE = 303
     <statement>
ELSE
     <statement>
END IF
```

In this example different statements are executed depending on whether mode is 301, 302, or 303, and if mode is none of them a fourth statement is executed.

### 4.5 The BEFORE Statement

The BEFORE statement is a "modifier" used to create an alternative condition as part of a WHEN (4.1.2), WHENEVER (4.2.2), or EVERY (4.3.2) construct. Please consult the referenced sections of this User's Guide for an explanation of how BEFORE is used.

The form of the BEFORE statement is:

```
      BEFORE  <singular_boolean>
```

where <singular_boolean> is an unarrayed component that may be evaluated as TRUE or FALSE. TIMELINER components are described below in Section 7.4.

### 4.6 The WITHIN Statement

The WITHIN statement is a "modifier" used to create an alternative condition as part of a WHEN (4.1.2), WHENEVER (4.2.2), or EVERY (4.3.2) construct. Please consult the referenced sections of this User's Guide for an explanation of how WITHIN is used.

The form of the WITHIN statement is:

```
      WITHIN  <singular_numeric>
```

where <singular_numeric> is an unarrayed numeric component that expresses a time interval in seconds. TIMELINER components are described below in Section 7.4.

### 4.7 The OTHERWISE Statement

The OTHERWISE statement is used as part of a modified WHEN construct to introduce statements that are executed instead of the normal statements in the event that the BEFORE or WITHIN condition arises before or at the same time as the condition given in the WHEN statement itself, as described in Section 4.1.2 of this User's Guide.

The form of the OTHERWISE statement is:

```
      OTHERWISE
```

The line must contain no additional material except comments.

### 4.8 The ELSE Statement

The ELSE statement is used as part of an IF construct to introduce statements to be executed if the original IF condition, or preceding ELSE IF conditions, fail -- as described in Section 4.4 above.

The ELSE statement may consist of the single word "ELSE", standing alone on a line. In this case the statements following the ELSE statement are executed if the <singular_boolean> specified in the IF statement, and in any ELSE IF statements that may form part of the construct, evaluate to be false.

The ELSE statement may alternatively be of the form:

```
      ELSE  IF  <singular_boolean>
```

where <singular_boolean> is any unarrayed component that may be evaluated as TRUE or FALSE. TIMELINER components are described below in Section 7.4.

In this case the statements following the ELSE statement are executed if the <singular_boolean> specified in the IF statement, and in any preceding ELSE IF statements, evaluate to be false, and if the <singular_boolean> in the present ELSE IF statement evaluates to true.

A particular IF construct may contain any number of ELSE IF statements, but at most one plain ELSE statement.

### 4.9 The END Statement

The END statement is used to conclude control constructs, namely the WHEN (4.1), WHENEVER (4.2), EVERY (4.3), and IF (4.4) constructs. Please consult the referenced sections of this User's Guide for an explanation of how END is used.

The form of the END statement is:

```
      END  [<construct_type>]
```

where <construct_type> is the single word WHEN, WHENEVER, EVERY, or IF. The <construct_type>, if included, must agree with the type of the construct being concluded.

### 4.10 The WAIT Statement

The WAIT statement is used to introduce a pause into a TIMELINER sequence. Only the particular sequence (or subsequence) containing the WAIT statement pauses.

The WAIT statement is of the form:

```
      WAIT  <singular_numeric>
```

where <singular_numeric> is an unarrayed numeric component that expresses a time interval in seconds. If the time interval is a variable, it will be evaluated only once, at the beginning of the wait. If the time interval is zero or negative, no pause will occur.

### 4.11 The CALL Statement

The CALL statement is used to invoke a subsequence, as described in Section 2.3 above. A CALL statement may be present in any TIMELINER sequence or subsequence within a bundle, and may be nested within a control construct.

The CALL statement is of the form:

```
CALL  <name>
```

When encountered, a CALL statement causes control to be transferred to the SUBSEQUENCE header statement that begins the subsequence that bears the specified <name>. The subsequence then executes. When the CLOSE statement that concludes the subsequence is encountered, control is returned to the sequence or subsequence that contains the CALL statement. Execution then continues at the statement following the CALL statement.

Because a subsequence, like all TIMELINER input, is simply a script to be acted out, and because the TIMELINER internal registers containing the status of a sequence (such as the line pointer) are owned by the calling sequence not the subsequence, all subsequences are "reentrant". That is, a subsequence may be called without worrying about whether the same subsequence may still be in execution due to another call from another sequence.

A called subsequence must lie within the same TIMELINER bundle as the sequence or subsequence that contains the CALL statement.

A control construct such as WHEN, WHENEVER, EVERY or IF must be completed within the same sequence or subsequence where it begins. That is, a control construct may not begin in the calling sequence (or subsequence) and end in the called subsequence, or vice versa.

## 5.0 ACTION STATEMENTS

This chapter describes the "action" statements that specify the actions to be taken under conditions specified by TIMELINER control statements.

The action statements implemented in the Ada-language version of TIMELINER at the present time are the LOAD, PRINT, SET, RESET, START, STOP, RESUME, and MESSAGE statements. The dummy action statement ACTION is also available. Further action statements will be added as the need arises.

### 5.1 The LOAD Statement

The LOAD statement is used to load data into a particular variable known to TIMELINER. A variable may be loaded from data provided as literals in the LOAD statement as well as from another variable.

The LOAD statement is normally used to initialize a variable at the start of a run, or to set a variable at some point during the run under conditions specified by means of TIMELINER control statements.

The LOAD statement has the form:

```
      LOAD  <name_of_variable>  =  <data>
```

where <name_of_variable> names any numeric, boolean, or character string variable known to TIMELINER.

The variable to be loaded may be arrayed in up to three dimensions. Subscripts may form part of the <name_of_variable>, in which case they lie inside parentheses, separated by commas. An asterisk subscript ("*") may be used to indicate that multiple array components are to be loaded.

The <data> portion of a LOAD statement consists of a component or list of components that are compatible in type with the variable to be loaded. That is, data to be loaded into a boolean variable must be of boolean type, data to be loaded into a numeric variable must be numeric, data to be loaded into a character string variable must be of character string type.

In addition, the data that appears in a LOAD statement must be compatible in size with the variable to be loaded. This means that the number of elements in the <data> field must either be one, or must be equal to the total number of elements in the variable to be loaded. If the variable is plural and the data contains only a single element, then that single element will be loaded into each element of the variable. If both the variable and the data are plural, then each element of the variable will be loaded into the corresponding element of the <data>. If both the variable and the data are plural, but their sizes do not match, then an error message will be issued during compilation.

The <data> field of a LOAD statement may consist of literals, other variables, combinations, defined components, or a list that mixes these types. Scale factors may be applied to loaded data by means of an arithmetic combination that multiplies data by some conversion constant.

Note that in earlier versions of TIMELINER the <data> field of a LOAD statement could consist only of literals, and the LOAD-FROM form of the statement was required to load a variable from another variable. This distinction is no longer present in TIMELINER and the LOAD-FROM statement no longer exists.

Here are some examples of valid LOAD statements:

```
      LOAD ABORT_FLAG = ON                -- ABORT_FLAG is boolean

      LOAD R =  +1.743883E+6, .1, .1      -- R is 3-vector

      LOAD DAP_MODE = "AUTO"              -- DAP_MODE is char string

      LOAD FD3(1,*,*) =  1,  2,  3,  4,
                         5,  6,  7,  8,
                         9, 10, 11, 12,
                        13, 14, 15, 16    -- FD3 is 4x4x4 numeric array

      LOAD V(1) = M_TO_FT * 363.3         -- V is vector scaled in f/s

      LOAD T1 = 23:23:23:23               -- data converted to seconds

      LOAD FD1 = RAD_TO_DEG * .01744            -- all components loaded to 1

      LOAD GUID_ACTIVE = SIM_FLAG OR NOT ABORT_FLAG

      LOAD FD2(1,2) = FD2(2,1) + FD2(2,2)

      LOAD DAP_MODE = GUID_MODE

      LOAD H = SQRT (A**2 + B**2)
```

Note that the LOAD capability is equivalent to an assignment statement. The LOAD statement may be used to perform algebraic computations in the TIMELINER language. (The last example above computes the hypotenuse of a right triangle.)

## 5.2 The PRINT Statement

The PRINT statement is used to print the contents of any variable known to TIMELINER.

The PRINT statement has the form:

```
      PRINT  <name_of_variable>
```

where <name_of_variable> names any numeric, boolean, or character string variable known to TIMELINER.

When it is executed the PRINT statement causes the named variable to be printed. An effort is made to print the variable in an easy-to-read format.

## 5.3 The SIGNAL Statement

The SIGNAL statement is used to set an "event". In the present implementation an event is a two-valued object that can be waited for and used in CIFO scheduling statements. (Note that the SIGNAL statement was formerly the SET statement.)

The SIGNAL statement is of the form:

```
SIGNAL  <name_of_event>
```

where <name_of_event> names any unarrayed event known to TIMELINER. Events, like variables, are made known to TIMELINER by being entered into the variable list, as described below in Section 9.2.

When the SIGNAL statement is executed the named event will be placed in an ON state.

## 5.4 The CLEAR Statement

The CLEAR statement is used to reset an "event". In the present implementation an event is a two-valued object that can be waited for and used in CIFO scheduling statements. (Note that the CLEAR statement was formerly the RESET statement.)

The CLEAR statement is of the form:

```
CLEAR  <name_of_event>
```

where <name_of_event> names any unarrayed event known to TIMELINER. Events, like variables, are made known to TIMELINER by being entered into the variable list, as described below in Section 9.2.

When the CLEAR statement is executed the named event will be placed in an OFF state.

## 5.5 The START Statement

The START statement is used to activate a sequence that lies within the same bundle.

The START statement is of the form:

```
START  <name_of_sequence>
```

where <name_of_sequence> is an alphanumeric word that corresponds to the name on the SEQUENCE header statement of another sequence within the same bundle.

If the named sequence is presently inactive, the START statement causes its statement pointer to be re-initialized to the first statement in the sequence and the sequence is then activated. If the named bundle is currently active the START statement will have no effect.

### 5.6 The STOP Statement

The STOP statement is used to deactivate a sequence that lies within the same bundle.

The STOP statement is of the form:

```
STOP  <name_of_sequence>
```

where <name_of_sequence> is an alphanumeric word that corresponds to the name on the SEQUENCE header statement of another sequence within the same bundle.

If the named sequence is presently active, the STOP statement causes it to be deactivated. If the named bundle is currently inactive the STOP statement will have no effect.

### 5.7 The RESUME Statement

The RESUME statement is used to activate a sequence that lies within the same bundle. RESUME differs from START by starting the sequence wherever it was stopped, rather than starting it at its first statement.

The RESUME statement is of the form:

```
RESUME  <name_of_sequence>
```

where <name_of_sequence> is an alphanumeric word that corresponds to the name on the SEQUENCE header statement of another sequence within the same bundle.

If the named sequence is presently inactive, the RESUME statement causes it to be activated. The sequence's statement pointer is not changed, so the first statement executed will be the statement at which the sequence was stopped. If the named bundle is currently active the RESUME statement will have no effect.

### 5.8 The MESSAGE Statement

The MESSAGE statement saves character string information provided at compilation such that it is available during execution. This capability can be used to supply informative material or to request manual actions.

The MESSAGE statement is of the form:

```
MESSAGE  <character_component>
```

where <character_component> is any component of character string type, including a character literal enclosed by double quotation marks.

In existing implementations the character information provided in a MESSAGE statement is made available during execution by printing it as part of the statement by statement output that is provided.

## 6.0 NON-EXECUTABLE STATEMENTS

The following non-executable statement is available. This statement has a "scope". That is, a non-executable statement placed at the top of a bundle has an effect that extends across all the sequences within the bundle. A non-executable statement placed at the top of a sequence or subsequence has effect only within the sequence or subsequence.

Note that future versions of TIMELINER will contain a DECLARE statement used to create variables for local use within a bundle, sequence, or subsequence.

### 6.1 The DEFINE Statement

The DEFINE statement is used to create a name that references some other component.

The DEFINE statement is of the form

```
      DEFINE  <name>  AS  <component>
```

where <name> is an alphanumeric word without embedded blanks, and <component> is any legal component.

Examples of the DEFINE statement are

```
      DEFINE  ALTITUDE  AS  R_MAGNITUDE - EARTH_RADIUS
      DEFINE  TEMP_RANGE  AS  100..125
```

The DEFINE statement creates a name. Within the scope of the statement, any reference to the name has the same effect as a reference to the component to which the name refers.

A DEFINE statement must be placed before the first executable statement in a bundle, sequence, or subsequence. The scope of the DEFINE statement depends upon its placement. A DEFINE placed at the top of a bundle creates a name that is visible within all the sequences and subsequences of the bundle. A DEFINE statement placed at the top of a particular sequence or subsequence creates a name that is visible throughout that sequence or subsequence.

### 6.2 The DECLARE Statement

The DECLARE statement is used to allocate a variable for use internal to a TIMELINER bundle, sequence, or subsequence.

The DECLARE statement is of the form

```
      DECLARE  <name>  BOOLEAN | NUMERIC | CHARACTER[(numeric_lit)]
```

where <name> is an alphanumeric word without embedded blanks. The type of the declared variable is designated by the word BOOLEAN. NUMERIC, OR CHARACTER. One and only one of these words must be present. If a singular, integer literal, enclosed by parentheses, follows the type designation, an array with the specified number of elements is created. Otherwise an unarrayed variable of the desired type is created.

Examples of the DEFINE statement are

```
     DECLARE  PUMP_ON  BOOLEAN          creates a single boolean
     DECLARE SPEED_OF_PUMP NUMERIC      creates a single numeric
     DECLARE  R_VECTOR  NUMERIC(3)      creates array of 3 numerics
     DECLARE FCS_STATUS CHARACTER(10)   creates a 10-character string
```

A DECLARE statement must be placed before the first executable statement in a bundle, sequence, or subsequence block. The scope of the DECLARE statement depends upon its placement. A DECLARE placed at the top of a bundle creates a name that is visible within all the sequences and subsequences of the bundle. A DECLARE statement placed at the top of a particular sequence or subsequence creates a name that is visible throughout that sequence or subsequence.

Arrayed internal variables may be referred to as a whole. (The ability to use subscripted references to particular elements of an arrayed internal variable will be implemented in a later release.)

Internal variables are initialized to null values at the start of a run (or upon the "installation" of a bundle) — boolean internal variables are set to FALSE, numeric internal variables to zero, and character internal variables to blank. Internal variables thereafter retain whatever value they were previously given. Thus an internal variable may not have its null value upon the second or later invocation of a block.

Note that when an internal variable is declared, static storage is established for that variable. The variable is not dynamically allocated when the bundle, sequence or subsequence containing the declaration is invoked. Note that if a subsequence is called recursively or reentrantly, the subsequence's internal variables are shared by the multiple invocations of the subsequence.

## 7.0 SYNTACTICAL ELEMENTS

This chapter describes the syntactical elements that are available for forming statements. Such syntactical elements include reserved words, operators, and components.

### 7.1 Reserved Words

This section summarizes all the reserved words that go into the formation of statements and components. For information on how each reserved word is used see the section whose number is given in parentheses.

| | |
|---|---|
| BUNDLE | blocking statement type (3.1) |
| SEQ | blocking statement type (3.2) |
| SEQUENCE | blocking statement type (3.2) |
| SUBSEQ | blocking statement type (3.3) |
| SUBSEQUENCE | blocking statement type (3.3) |

| | |
|---|---|
| CLOSE | blocking statement type (3.4) |
| ACTIVE | SEQUENCE-statement keyword (3.2) |
| INACTIVE | SEQUENCE-statement keyword (3.2) |
| WHEN | control statement/construct type (4.1) |
| WHENEVER | control statement/construct type (4.2) |
| EVERY | control statement/construct type (4.3) |
| IF | control statement/construct type (4.4) |
| BEFORE | control statement type (4.5) |
| WITHIN | control statement type (4.6) |
| OTHERWISE | control statement type (4.7) |
| ELSE | control statement type (4.8) |
| END | control statement type (4.9) |
| WAIT | control statement type (4.10) |
| CALL | control statement type (4.11) |
| LOAD | action statement type (5.1) |
| PRINT | action statement type (5.2) |
| SIGNAL | action statement type (5.4) |
| CLEAR | action statement type (5.5) |
| START | action statement type (5.6) |
| STOP | action statement type (5.7) |
| RESUME | action statement type (5.8) |
| MESSAGE | action statement type (5.9) |
| DEFINE | non-executable statement type (6.1) |
| AS | DEFINE-statement keyword (6.1) |
| DECLARE | non-executable statement type (6.2) |
| BOOLEAN | DECLARE-statement keyword (6.2) |
| NUMERIC | DECLARE-statement keyword (6.2) |
| CHARACTER | DECLARE-statement keyword (6.2) |
| AND | logical operator (7.3.1) |
| OR | logical operator (7.3.1) |
| NOT | logical operator (7.3.1) |
| IN | range operator (7.3.6) |
| OUTSIDE | range operator (7.3.6) |
| MOD | arithmetic operator (7.3.4) |
| ON | boolean literal (7.4.1.1) |
| OFF | boolean literal (7.4.1.1) |
| TRUE | boolean literal (7.4.1.1) |
| FALSE | boolean literal (7.4.1.1) |
| DEG_TO_RAD | built-in constant (7.4.2) |
| RAD_TO_DEG | built-in constant (7.4.2) |
| FT_TO_M | built-in constant (7.4.2) |
| M_TO_FT | built-in constant (7.4.2) |
| FT_TO_NM | built-in constant (7.4.2) |
| NM_TO_FT | built-in constant (7.4.2) |
| G_TO_FPS2 | built-in constant (7.4.2) |
| FPS2_TO_G | built-in constant (7.4.2) |
| LBM_TO_KG | built-in constant (7.4.2) |
| KG_TO_LBM | built-in constant (7.4.2) |
| SLUG_TO_KG | built-in constant (7.4.2) |
| KG_TO_SLUG | built-in constant (7.4.2) |
| LBF_TO_N | built-in constant (7.4.2) |
| N_TO_LBF | built-in constant (7.4.2) |

| | |
|---|---|
| ABS | built-in function (7.4.3) |
| SQRT | built-in function (7.4.3) |
| SIN | built-in function (7.4.3) |
| COS | built-in function (7.4.3) |
| TAN | built-in function (7.4.3) |
| ARCSIN | built-in function (7.4.3) |
| ARCCOS | built-in function (7.4.3) |
| ARCTAN | built-in function (7.4.3) |

## 7.2 Names

Names are created by the following statements:

| | |
|---|---|
| BUNDLE | name of bundle (Section 3.1) |
| SEQUENCE | name of sequence (Section 3.2) |
| SUBSEQUENCE | name of subsequence (Section 3.3) |
| DEFINE | name of defined component (Section 6.1) |
| DECLARE | name of internal variable (Section 6.2) |

Names are a string of characters chosen by the writer of the script. Names must be unique within their scope. Bundle names should be unique within the set of available bundles. Sequence and subsequence names, and names created by a DEFINE or DECLARE statement appearing at the top of a bundle must be unique within the bundle. Names created by a DEFINE or DECLARE statement within a sequence or subsequence must be unique within their sequence or subsequence.

Names must begin with a letter of the alphabet, and must avoid using any symbols that are used to form operators (see Section 7.3). Names may not contain blanks, but underscores are permitted. Names are limited in length to an implementation-dependent maximum, which is currently set at 40 characters. A name may not be one of the reserved words listed in the previous section.

## 7.3 Operators

This section summarizes the operators that are used in the formation of components. Somewhat arbitrarily the operators are grouped into several categories.

The order of precedence of operators is as follows, ranging from the operators that create the loosest associations to those that create the tightest:

| | | |
|---|---|---|
| AND | (Note 1) | logical operator (7.3.1) |
| OR | (Note 1) | logical operator (7.3.1) |
| = | | equality operator (7.3.2) |
| /= | | equality operator (7.3.2) |
| < | | comparison operator (7.3.3) |
| <= | | comparison operator (7.3.3) |
| > | | comparison operator (7.3.3) |
| >= | | comparison operator (7.3.3) |
| , | | list operator (7.3.5) |
| IN | (Note 1) | range operator (7.3.6) |
| OUTSIDE | (Note 1) | range operator (7.3.6) |

|  |  |  |
|---|---|---|
| .. |  | range operator (7.3.6) |
| + | (Note 2) | arithmetic operator (7.3.4) |
| - | (Note 2) | arithmetic operator (7.3.4) |
| * |  | arithmetic operator (7.3.4) |
| / |  | arithmetic operator (7.3.4) |
| MOD | (Note 1) | arithmetic operator (7.3.4) |
| ** |  | arithmetic operator (7.3.4) |
| NOT | (Note 1) | logical operator (7.3.1) |
| + | (Note 3) | arithmetic (sign) operator (7.3.4) |
| - | (Note 3) | arithmetic (sign) operator (7.3.4) |

Note 1: This operator will be recognized only if it is preceded by a blank or a close-parenthesis, and anteceded by a blank or open-parenthesis.

Note 2: If preceded by an "E" and anteceded by a numeral the plus and minus symbols are interpreted as forming part of an exponent attached to a numeric literal.

Note 3: If the parser finds no components to their left, the plus and minus symbols are interpreted as unary operators that modify the numeric component lying to their right.

### 7.3.1 Logical Operators

The TIMELINER operators AND, OR, and NOT are known as "logical" operators. These operators operate only on boolean components.

*NOT*

The NOT operator is a unary operator that modifies the sense of the singular or plural boolean component lying to its right. The result of the operator is a component of equal size with the logical sense of each element reversed.

If the NOT operator has any component to its immediate left (with no other operator in between) an error message will be issued at compile time.

Here are examples of proper and improper uses of the NOT operator:

```
ALT NOT = 12345                 ==> illegal
ALT /= 12345                    ==> legal
NOT (ALT = 12345)               ==> legal
```

*AND, OR*

The AND and OR operators are binary operators that create a component of <boolean_combo> type (Section 7.4.4.1) by making the appropriate logical combination of the components to the left and right.

The components to the left and right must both be of boolean type and a compiler error is generated if they are not.

The components to the left and right of the operator must be compatible in size. If the size is equal the resulting component will be the appropriate logical combination of the corresponding elements of the left and right components. If one component is singular and the other has more than one element, the resulting plural component is the appropriate logical combination of each element of the plural component with the singular component. If both components are plural, but their size is not equal, an error message is generated by the compiler.

For example

```
(ON, OFF, ON)  AND  (ON, ON, OFF)     ==>  (ON, OFF, OFF)
(ON, OFF, ON)  OR  OFF                ==>  (ON, OFF, ON)
(ON, OFF, ON)  AND  (ON, OFF)         ==>  illegal
```

A compiler error is generated if the components to the left and right of the AND or OR operator are both plural, but have an unequal number of elements.

### 7.3.2 Equality Operators

The TIMELINER equality operators are as follows:

| | |
|---|---|
| = | equals |
| /= | not equals |

The equality operators are binary operators that create a component of <boolean_combo> type (Section 7.4.4.1). The resulting component is always singular.

The components to the left and right of an equality operator must be of the same type, whether boolean, numeric, or character. A compiler error is generated if they are not.

The "=" operator returns the value TRUE if every element of the component on one side equals the corresponding element on the other side. Otherwise FALSE is returned.

If one of the components being combined is plural and the other is singular, TRUE is returned if every element of the plural component is equal to the singular component. Otherwise FALSE is returned.

If the components to the left and right are both plural, but of unequal sizes, an error message is generated by the compiler.

The "/=" operator returns the reverse of what the "=" operator would return.

### 7.3.3 Comparison Operators

The TIMELINER comparison operators are as follows:

| | |
|---|---|
| < | less than |
| <= | less than or equals |
| > | greater than |
| >= | greater than or equals |

The comparison operators are binary operators that create a component of <boolean_combo> type (Section 7.4.4.1). The resulting component is always singular.

The components to the left and right of a comparison operator must both be of numeric type, and must both be singular. A compiler error is generated if either is not.

A comparison operator returns TRUE if the magnitude relationship expressed by the operator corresponds to the magnitude relationship of the components to the left and right. Otherwise FALSE is returned.

### 7.3.4 Arithmetic Operators

The TIMELINER arithmetic operators are as follows:

| | |
|---|---|
| + | addition |
| - | subtraction |
| * | multiplication |
| / | division |
| MOD | modulo |
| ** | exponentiation |

The arithmetic operators are binary operators that create a component of <numeric_combo> type (Section 7.4.4.2). The resulting component has a size (arrayness) equal to the larger of the components to the left and right of the operator.

The components to the left and right of an arithmetic operator must both be of numeric type, and a compiler error is generated if either is not.

The components to the left and right of the operator must be compatible in size. If the size is equal the resulting component will be the appropriate arithmetic combination of the corresponding elements of the left and right components. If one component is singular and the other has more than one element, the resulting plural component is the appropriate arithmetic combination of each element of the plural component with the singular component. If both components are plural, but their size is not equal, an error message is generated by the compiler.

Note that the plus and minus operators may be used in other ways than as simple binary operators. A plus or minus that has no component to its immediate left is interpreted as a sign modifying the component to its right. The resulting component types in these cases are <numeric_pos> and <numeric_neg>. Furthermore, a plus or minus operator preceded, without intervening blanks, by the character "E" and anteceded, without intervening blanks, by any numeral from 0 to 9 is interpreted as the sign of an exponent within a numeric literal expressed in scientific format.

### 7.3.5 List Operators

The TIMELINER "list" operator is the comma.

A list is formed by a series of components separated by commas. The components making up the list may be of any size, and the size of the resulting list component is the sum of the sizes of the constituent components.

If all the components within the list are of boolean type, the resulting component is of the type <boolean_list>. If all the components within the list are of numeric type, the resulting component is of the type <numeric_list>. If all the components within the list are of character string type, the resulting component is of the type <cstring_list>.

If the components within the list are of disparate type, the resulting component is of the type <mixed_list>. However, there is no context in which the <mixed_list> component is legal, and therefore a compiler error will be generated.

Note that a special case of a numeric list component is the "subscript list". A subscript list, enclosed in parentheses, may be used to designate a particular multi-dimensional array element A subscript list is recognized by its context. A subscript list must consist of components of singular numeric type, or of <wild_card> type (expressed as "*"), or of <range_fixed> type.

### 7.3.6 Range Operators

The TIMELINER range operators are as follows:

| | |
|---|---|
| .. | "to" (creates a range) |
| IN | value to the left inside the range on the right |
| OUTSIDE | value to the left outside the range on the right |

The range operator ".." is used to create a range component. The component is of type <range_fixed> if the components on both sides of the operator are numeric literals. Otherwise the resulting component is of type <range_variable>.

The range operators IN and OUTSIDE are used to compare a singular numeric component (on the left) to a range component to the right. The resulting component is of type <boolean_combo>, and is always singular.

A compiler error is generated if the component to the left is not of numeric type, or if it is plural. A compiler error is generated if the component to the right is not a range component.

For the IN operator, the value TRUE is returned if the numeric component to the left is <= the left-hand end of the range and <= the right-hand end of the range. Otherwise FALSE is returned. For the OUTSIDE operator the result is opposite.

Here is an example of a TIMELINER statement that uses range operators:

```
WHEN  GMT  IN  11:00:00..11:10:00
```

The WHEN condition passes if Greenwich mean time is between eleven and ten past eleven.

## 7.4 Components

Throughout this User's Guide reference has been made to "components". As used in this document the word "component" is simply a general way of referring to certain syntactical elements that recur in various TIMELINER statements.

Components have several attributes: type, size, and variety.

### Component Type

Most components fall into one of three major types. These types are boolean, numeric, and character string.

Boolean components are components that can be evaluated to obtain one or more booleans. Numeric components are components that can be evaluated to obtain one or more "numbers". Character components are components that can be evaluated to obtain a character string.

There are a few components, such as ranges, that do not fall neatly into the three major types. These are discussed separately.

### Component Size

A component's "size" is the number of elements formed by the evaluated component. When they are evaluated all components are considered to be one dimensional arrays with one or more elements. Thus "size" is equivalent to the "arrayness" of the component.

In this document the components <singular_boolean> and <singular_numeric> are sometimes referred to. What is meant by such a reference is a component of the boolean type, or numeric type, that has only one element -- an arrayness of one.

### Component Variety

The "variety" of a component may be one of the following:

literal
A fixed value of the appropriate type that is established at compile time. Must not be loaded by a LOAD statement.

variable
An arrayed or unarrayed variable of boolean, numeric, or string type. A variable may be loaded using the LOAD statement.

constant
A built-in constant of numeric type. Cannot be loaded.

combination
A value created by combining two other components using an operator of appropriate type. Cannot be loaded.

definition
A value of any component type that has been given its own name by means of a DEFINE statement (Section 6.1). Allowed usage depends upon underlying component type.

list
A series of values created by writing some number of components separated by commas. Cannot be loaded.

range A two-valued component formed by linking two numeric components by means of the range operator "..". Cannot be loaded.

The following table summarizes the component varieties available in TIMLINER:

| VARIETY | BOOLEAN | NUMERIC | CHARACTER | OTHER |
|---|---|---|---|---|
| LITERAL | <boolean_true><br><boolean_false> | <num_scal_lit><br><num_ntgr_lit><br><num_time_lit> | <cstring_lit> | |
| SIMULATION VARIABLE | <boolean_var> | <numeric_var> | <character_var> | |
| INTERNAL VARIABLE | <bool_int_var> | <num_int_var> | <char_int_var> | |
| CONSTANT | | <numeric_const> | | |
| FUNCTION | | <numeric_funct> | | |
| COMBINATION | <boolean_combo><br><boolean_not> | <numeric_combo><br><numeric_neg><br><numeric_pos> | | |
| DEFINITION | <boolean_def> | <numeric_def> | <cstring_def> | <range_def> |
| LIST | <boolean_list><br><bool_data_list> | <numeric_list><br><num_data_list><br><subscript_list> | <cstring_list><br><cstr_data_list> | <mixed_list> |
| RANGE | | <range_fixed><br><range_variable><br><wild_card> | | |

The component varieties summarized in this table are described in the following sections.

### 7.4.1 Literal Components

Literal components may be of boolean, numeric, or character type, as described in the following subsections.

#### 7.4.1.1 Boolean Literals

The boolean literals supported by TIMELINER are as follows:

<boolean_true> TRUE *or* ON

| | |
|---|---|
| <boolean_false> | FALSE *or* OFF |

### 7.4.1.2 Numeric Literals

The numeric literals supported by TIMELINER are any alphanumeric strings that can be converted to an 8-byte floating point number or 4-byte integer by the GET routines provided as part of the Ada package TEXT_IO.

In addition TIMELINER supports several forms that are not accepted by TEXT_IO, as follows:

* Numbers that begin with a decimal point, or with a sign followed immediately by a decimal point.

* Numbers that use the familiar DDD:HH:MM:SS.SS format often used to express a time in terms of days, hours, minutes and seconds. Numbers in the formats HH:MM:SS.SS and MM:SS.SS are also accepted. Note that the various fields are not magnitude checked. Thus the forms 96:00:12.1 and 4:00:00:12.1 are both legal, and they are equivalent. The "seconds" field may or may not contain a decimal point. The days, hours, and minutes fields must not contain a decimal point. A time-format literal is converted to seconds for use.

A numeric literal is filed as <num_ntgr_lit> if it is written in the form of an integer, <num_scal_lit> if it contains a decimal point, or <num_time_lit> if it is expressed in time format, as indicated by the presense of colons. However this distinction is maintained only to allow the literal to be printed in an appropriate style during execution. TIMELINER stores all numeric literals as 8-byte floating point numbers.

Here are some examples of valid numeric literals (the right hand column indicates how the number is stored internally):

| | |
|---|---|
| 1000 | 1.00000000000000E+03 |
| -.75 | -7.50000000000000E-01 |
| 1.23456789012345E+67 | 1.23456789012345E+67 |
| 11:11 | 6.71000000000000E+02 |
| 364:23:59:59.999 | 3.15359999990000E+07 |
| 1_222_333_444 | 1.22233344400000E+09 |
| 2#11111111# | 2.55000000000000E+02 |
| 16#AA# | 1.70000000000000E+02 |
| -9999999999999995.0 | -1.00000000000000E+16 |

A numeric literal always has a size of one. A series of numeric literals separated by commas is interpreted as a list (Section 7.3.5) of numeric literals.

### 7.4.1.3 Character Literals

A character string literal is any series of characters enclosed by single or double quotation marks. A character string literal may contain blanks. It may contain any number of quotation marks of the other type, but none of the same type.

### 7.4.2 Simulation Variable Components

Simulation variable components are components that name a variable that is defined within the simulation to which TIMELINER is attached. Such variables must be present in common areas (Ada package specifications), and information describing each variable must be entered into the "variable list" that must form part of both the TIMELINER compiler and the TIMELINER executor.

Simulation variables may be boolean, numeric, character string, or event variables. They may be arrayed in up to three dimensions. Each element must belong to one of the predefined basic data types. Enumeration-type variables are dealt with as integers. The user may refer to them in a script as enumeration variables by using the DEFINE capabilty to create names standing for integer values.

The means by which simulation variables are entered into the variable list is detailed in Section 9.2.

### 7.4.3 Internal Variable Components

Internal variable components are local variables creates by the use of the DECLARE statement described in Section 6.2.

Internal variables may be of boolean, numeric or character type, and may be arrayed in up to one dimension. Internal variables declared at the top of a bundle may be referred to by any sequence or subsequence in the bundle. Internal variables declared within a sequence or subsequence may be used only within that sequence or subsequence.

Arrayed internal variables may be referred to as a whole. Subscripted references are not permitted. (This capability will be added in a later release.)

When an internal variable is declared, static storage is established for that variable. The variable is not dynamically allocated when the bundle, sequence or subsequence containing the declaration is invoked. Note that if a subsequence is called recursively or reentrantly, the subsequence's internal variables are shared by the multiple invocations of the subsequence.

Internal variables are initialized to null values at the start of a run — boolean internal variables are set to FALSE, numerics to zero, and character internal variables to blank.

### 7.4.4 Constant Components

TIMELINER contains a number of built-in constants which are available for use in converting from one system of units to another. The constants supported by TIMELINER are the following:

| | |
|---|---|
| DEG_TO_RAD | converts degrees to radians |
| RAD_TO_DEG | converts radians to degrees |

| | |
|---|---|
| FT_TO_M | converts feet to meters |
| M_TO_FT | converts meters to feet |
| FT_TO_NM | converts feet to nautical miles |
| NM_TO_FT | converts nautical miles to feet |
| G_TO_FPS2 | converts G's to feet/sec/sec |
| FPS2_TO_G | converts feet/sec/sec to G's |
| LBM_TO_KG | converts pounds mass to kilograms |
| KG_TO_LBM | converts kilograms to pounds mass |
| SLUG_TO_KG | converts slugs to kilograms |
| KG_TO_SLUG | converts kilograms to slugs |
| LBF_TO_N | converts pounds force to newtons |
| N_TO_LBF | converts newtons to pounds force |

Only constants of numeric type are defined, and these are filed as a component of the type <numeric_const>. All such constants are singular, i.e., their size (arrayness) is one.

### 7.4.5 Function Components

TIMELINER contains a number of built-in functions. The functions supported by TIMELINER are the following:

ABS
SQRT
SIN
COS
TAN
ARCSIN
ARCCOS
ARCTAN

Only built-in functions of numeric type are defined, and these are filed as a component of the type <numeric_funct>. All such functions are singular, i.e., their size (arrayness) is one.

### 7.4.6 Combination Components

Combination components are components created by combining other components by means of the operators listed above in Section 7.3. Combinations may be of boolean or numeric type. No character string combinations are supported.

#### 7.4.6.1 Boolean Combinations

Boolean combinations are combinations formed by the use of the operators

AND OR NOT = /= < <= > >=

The combination formed by the NOT operator is filed as <boolean_not>. It is of the form:

NOT <boolean_component>

Where <boolean_component> may be any singular or plural boolean component. When evaluated the <boolean_not> is a component of the same size as the <boolean_component>, each of whose elements is the opposite of the corresponding element of the <boolean_component>.

The other forms of the component <boolean_combo> are of the form:

<component> <operator> <component>

In the case of the AND operator and the OR operator, the two components must be of boolean type, and if both are plural their size must be equal. The result is a <boolean_combo> component whose size is equal to the size of the plural component(s), if any.

In the case of the "equals" and "not equals" operators, the two components may be of any type, and if both are plural their size must be equal. The result is a <boolean_combo> component of size one. In the case of the "equals" operator the result is TRUE when the two components are equal element by element, or when each element of a plural component equals a singular component. In the case of the "not equals" operator the result is FALSE under the same conditions.

In the case of the magnitude operators, the two components must be of singular numeric type. The result is a <boolean_combo> component of size one, which is TRUE when the stated magnitude relationship between the two numeric components is true.

### 7.4.6.2 Numeric Combinations

Numeric combinations are combinations formed by the use of the addition (plus), subtraction (minus), multiplication, division, modulo, and exponentiation operators written as follows:

+ - * / MOD **

The combination formed by the plus and minus operators is filed as <numeric_pos> or <numeric_neg> if it is of the form:

<sign> <numeric_component>

Where <numeric_component> may be any singular or plural numeric component. When evaluated the result is a numeric component each of whose elements is the positive or negative of the corresponding element of the <numeric_component>.

The other configurations of the component <numeric_combo> are of the form:

<numeric_component> <operator> <numeric_component>

The two components must be of numeric type, and if both are plural their sizes must be equal. The resulting component is a <numeric_combo> whose size is equal to the size of the plural component(s), if any. Each element of the resulting component is created by performing the stated arithmetic operation on the corresponding elements of the two components, or on each element of the plural component with respect to the singular component.

The following are examples of numeric combinations:

```
(1, 2, 3)  +  (3, 2, 1)     ==>   (4, 4, 4)
(1, 2, 3)  /  2             ==>   (0.5, 1.0, 1.5)
(1, 2, 3)  *  (1, 2)        ==>   illegal
```

### 7.4.7 Definition Components

Components may be formed by means of the DEFINE statement described above in Section 6.1. A definition component is classified as a <boolean_def>, <numeric_def>, <cstring_def> or <range_def> depending on whether the component stated in the DEFINE statement is of boolean, numeric, character string, or range type. The size of the definition component is equal to the size of the referenced component.

### 7.4.8 List Components

A list is formed by a series of singular or plural components separated by commas. It may optionally be enclosed by parentheses.

A list is classed as a <boolean_list> if every member of the list is a component of boolean type. A list is classed as a <numeric_list> if every member of the list is a component of numeric type. A list is classed as a <cstring_list> if every member is a component of character type.

A list appearing as a subscript to a multi-dimensional array is classified as a <subscript_list>. Every element of a subscript list must be a <singular_numeric>, a <range_fixed>, or a <wild_card>.

A list is classed as a <mixed_list> if the members are of more than one type. However, the <mixed_list> component has no legal use in TIMELINER so the creation of a mixed list will result in a compiler error message.

The size of a list component is equal to the sum of the sizes of the member components.

Note that a character string list is equivalent to a concatenation of character strings.

### 7.4.9 Range Components

A range component is of the form:

```
<numeric_component>  ..  <numeric_component>
```

The numeric components used to form a range must be of size one. The resulting range component does not possess a size.

A range is filed as a <range_fixed> component if both numeric components are of numeric literal type, or numeric constant type, or of definition type where the defined component is of fixed or constant type. A range is filed as a <range_variable> if either of the numeric components is of any variable numeric type or function.

There is a special case of a range component known as a <wild_card>. A "wild card" is legal only within a subscript. The range implied is the range corresponding to the allowable subscripts for the particular dimension of the array.

## 8.0 TIMELINER ERROR MESSAGES

At compile-time TIMELINER issues error messages designed to facilitate the user's effort to create an executable TIMELINER script. TIMELINER error messages are called "cusses". This chapter describes the various cusses that may be issued.

TIMELINER makes an effort to keep going, even if there are many errors, in the hope of giving the user enough information to eliminate all errors in a single pass. There is no maximum number of errors that will cause TIMELINER to quit parsing the input stream.

The authors of TIMELINER would like to be told if you encounter cases where the error messages that are issued seem misleading, or where the presence of one error avoidably masks the existence of another.

In the headings below, the phrases given after the number represent the values of the enumeration-type used when invoking the "cuss" capability from the TIMELINER compiler..

The following error messages are issued by TIMELINER:

### Cuss 101 -- unrecognized_directive

The compile-time error message

```
Unrecognized DIRECTIVE statement:
```

is issued if a compiler directive statement (described in Section 2.7) cannot be recognized.

### Cuss 102 -- print_level_not_numeric

The compile-time error message

```
PRINT_LEVEL may only be set to a numeric literal
```

is issued if a "directive" statement intended to change TIMELINER's compile-time "print level" expresses the desired level as anything other than an integer literal..

**Cuss 121 -- too_many_blocks**

The compile-time error message

> Too many blocks (BUNDLE/SEQ/SUBSEQ) in script -- maximum is nbk

is issued if the number of bundles, sequences, and subsequences in the script exceeds the capacity of the TIMELINER tables. The number given in the cuss is the maximum number of bundles, sequences, and subsequences that can currently be compiled. This number can be increased by changing the parameter "nbk" in the module TL_DATA_COM and recompiling all TIMELINER modules. For a given TIMELINER script, the usage of the table space available for bundles, sequences, and subsequences is shown as part of the "file usage summary" printed at the end of the compile-time listing.

**Cuss 122 -- too_many_names**

The compile-time error message

> Too many names within current scope -- maximum is nnm

is issued if the number of names created by the BUNDLE, SEQUENCE and SUBSEQUENCE blocking statements, and by the DEFINE and DECLARE statements, exceed the capacity of the compilation tables. This limit is applied not to the total number of names created by a script, but to the number of names recognized within a given scope. The allowed number of names is set by the parameter "nnm" defined in the module TL_DATA_COM.

**Cuss 123 -- too_many_stats**

The compile-time error message

> Too many statements in script -- maximum is nst

is issued if the number of statements in a BUNDLE exceeds a maximum set by the parameter "nst" defined in the module TL_DATA_COM.

**Cuss 124 -- too_much_comp_data**

The compile-time error message

> Too much component data in script -- maximum is ncd

is issued if the amount of component data in a BUNDLE exceeds a maximum set by the parameter "ncd" defined in the module TL_DATA_COM.

**Cuss 125 -- too_many_numeric_lits**

The compile-time error message

> Too many numeric literals in script -- maximum is nnl

is issued if the number of numeric literals (defined in Section 7.4.1.2) exceeds the capacity of the TIMELINER tables. The number given in the cuss is the maximum number of numeric literals that can currently be compiled. This number can be increased by changing the parameter "nnl" in the module TL_DATA_COM and recompiling all TIMELINER modules. For a given TIMELINER script, the usage of the table space available for numeric literals is shown as part of the "file usage summary" printed at the end of the compile-time listing. Use of the "optimize" compiler directive (Section 2.7) may reduce the number of numeric literals required by a script.

**Cuss 126 -- too_many_character_lits**

The compile-time error message

> Too many character string literals in script -- maximum is ncl

is issued if the number of characters devoted to string literals (defined in Section 7.4.1.3) exceeds the capacity of the TIMELINER tables. Character literal space is used for the storage of (1) strings enclosed by quotation marks (Section 7.4.1.3) , and (2) names created by BUNDLE (3.1), SEQUENCE (3.2), SUBSEQUENCE) (3.3), DEFINE (6.1) and DECLARE (6.2) statements. The number given in the cuss is the maximum number of characters that can currently be digested. This number is given by parameter "ncl" in the module TL_DATA_COM. The usage of character literal table space is shown as part of the "file usage summary" printed at the end of the compile-time listing. Use of the "optimize" compiler directive (Section 2.7) may reduce the number of characters required by a script.

**Cuss 131 -- too_many_finishers**

The compile-time error message

> There are too many CLOSE and/or END statements

is issued if the number of blocking statements used to conclude BUNDLES, SEQUENCES, and/or SUBSEQUENCES is greater than the number of blocking statements used to initiate them. Each CLOSE statement must have a corresponding BUNDLE, SEQUENCE, or SUBSEQUENCE. This error is also issued if the number of END statements encountered is greater than the number of WHEN, WHENEVER, EVERY, and IF constructs in effect. Each END statement must have an associated WHEN, WHENEVER, EVERY, or IF construct.

**Cuss 141 -- statement_too_long**

The compile-time error message

Statement is too long -- maximum length is nls

is issued if the number of characters in a TIMELINER statement exceeds the maximum permitted, as determined by the parameter "nls" defined in TL_DATA_COM. A TIMELINER statement may continue onto any number of lines until this limit is reached.

**Cuss 142 -- statement_not_recognized**

The compile-time error message

Statement type not recognized

is issued if a statement is encountered during a compiler compilation whose type cannot be recognized. (NOTE: If this error message occurs please bring it to the attention of the author.)

**Cuss 143 -- statement_not_implemented**

The compile-time error message

Statement type not implemented in this version:

is issued if the statement of the type indicated by the word printed after the colon is not implemented in the present version of TIMELINER. This cuss is used in cases where the statement type is legal in some other version of the TIMELINER language. (For example, in this version of TIMELINER the SET and COMMAND statements, which are legal in the UIL version, are not implemented.)

**Cuss 151 -- name_too_long**

The compile-time error message

The following name is too long (max length is nwl):

is issued if the number of characters used to represent the specified name exceeds the maximum defined by the parameter "nwl" in the module TL_DATA_COM. This limit applies to names created by the BUNDLE, SEQUENCE, SUBSEQUENCE, DECLARE and DEFINE statements.

**Cuss 152 -- name_already_in_use**

The compile-time error message

> Name is already in use -- see statement

is issued if the name given in a BUNDLE, SEQUENCE, SUBSEQUENCE, DECLARE or DEFINE statement has already been created by another statement within the same scope.. The number of the statement where the conflicting name was created is printed as part of the cuss.

**Cuss 153 -- name_same_as_variable**

The compile-time error message

> Name conflicts with name of variable:

is issued if the name given in a BUNDLE, SEQUENCE, SUBSEQUENCE, DECLARE or DEFINE statement is the same as the name of any simulation variable that has been entered into the TIMELINER variable list, as described in Section 9.2 of this User's Guide. The conflicting name is printed after the colon.

**Cuss 161 -- no_block_open_at_close**

The compile-time error message

> No BUNDLE, SEQ, or SUBSEQ block has been opened yet in this script

is issued if a CLOSE statement is encountered prior to any BUNDLE, SEQUENCE, or SUBSEQUENCE statement. The correct organization of a TIMELINER script into hierarchical blocks is explained in Chapter 2 of this User's Guide.

**Cuss 162 -- end_with_close_blocker**

The compile-time error message

> Input stream must end with CLOSE statement

is issued if the last functional statement in the input stream is not a CLOSE statement. Since all TIMELINER statements must belong to some bundle the last statement must always be a CLOSE BUNDLE statement. The correct organization of a TIMELINER script into hierarchical blocks is explained in Chapter 2 of this User's Guide.

**Cuss 171 -- bundle_nested_too_deep**

The compile-time error message

> A BUNDLE may not be nested inside another block

is issued if a BUNDLE header statement is encountered when an existing BUNDLE has not been closed. Only one BUNDLE may be open at a time. The correct organization of a TIMELINER script into hierarchical blocks is explained in Chapter 2 of this User's Guide.

**Cuss 172 -- bundle_must_come_first**

The compile-time error message

> A BUNDLE statement must be the first functional statement

is issued if a BUNDLE statement is not the first functional statement encountered. Only blank lines, comments and directive statements are allowed before the BUNDLE statement. The first statement in any TIMELINER script is normally a BUNDLE header statement and the last is normally a CLOSE BUNDLE statement. The correct organization of a TIMELINER script into hierarchical blocks is explained in Chapter 2 of this User's Guide.

**Cuss 181 -- seq_nested_too_deep**

The compile-time error message

> A SEQUENCE may not be nested at this level

is issued if a SEQUENCE header statement is encountered when an existing SEQUENCE or SUBSEQUENCE has not been closed. Only one SEQUENCE or SUBSEQUENCE may be open at a time. The correct organization of a TIMELINER script into hierarchical blocks is explained in Chapter 2 of this User's Guide.

**Cuss 182 -- subseq_nested_too_deep**

The compile-time error message

> A SUBSEQUENCE may not be nested at this level

is issued if a SUBSEQUENCE header statement is encountered when an existing SEQUENCE or SUBSEQUENCE has not been closed. Only one SEQUENCE or SUBSEQUENCE may be open at a time. The correct organization of a TIMELINER script into hierarchical blocks is explained in Chapter 2 of this User's Guide.

**Cuss 191 -- no_seq_or_subseq_open**

The compile-time error message

> This statement inappropriate because no SEQ or SUBSEQ is open

is issued if any control or action statement is found outside the confines of a SEQUENCE or SUBSEQUENCE. The correct organization of a TIMELINER script into hierarchical blocks is explained in Chapter 2 of this User's Guide.

**Cuss 192 -- statement_nesting_too_deep**

The compile-time error message

> Statement nesting too deep -- maximum number of levels is nsnl

is issued if the depth of statement construct nesting exceeds the capacity of the TIMELINER tables. The number given in the cuss is the maximum number of levels of nesting that can currently be handled. This number may be increased by changing the parameter "nsnl" in the module TL_DATA_COM and recompiling all TIMELINER modules. For a given TIMELINER script, the usage of the depth available for construct nesting is shown as part of the "file usage summary" printed at the end of the compile-time listing.

**Cuss 201 -- no_construct_open**

The compile-time error message

> No construct open when END statement encountered --
> use CLOSE to terminate block

is issued if the user attempts to use an END statement to close a BUNDLE, SEQUENCE, or SUBSEQUENCE block. END ends constructs only. CLOSE closes blocks.

**Cuss 202 -- construct_open_at_close**

The compile-time error message

> Construct open when CLOSE statement encountered

is issued if a control construct such as WHEN, WHENEVER, EVERY or IF remains open when a CLOSE SEQUENCE or CLOSE SUBSEQUENCE statement is encountered. All constructs must be closed (using END) before the sequence or subsequence that contains them can be closed. The cuss probably means that the END statement of a construct has been omitted.

**Cuss 211 -- block_not_named**

The compile-time error message

```
BUNDLE, SEQ, or SUBSEQ statement must include a name
```

is issued if a BUNDLE, SEQUENCE or SUBSEQUENCE header statement (Sections 3.1-3.3) does not contain a name to be applied to the block being opened.

**Cuss 212 -- extraneous_material**

The compile-time error message

```
Statement contains extraneous material:
```

is issued if there is additional material appended to a statement. However, extraneous material is not detected in all cases.

**Cuss 221 -- close_mismatched**

The compile-time error message

```
CLOSE statement type does not correspond to type of open block:
```

is issued if the type indicated by the second word in a CLOSE statement does not match the type of the innermost block (BUNDLE, SEQUENCE, or SUBSEQUENCE) that is open. The material after the colon is the erroneous word.

**Cuss 222 -- end_mismatched**

The compile-time error message

```
END statement type does not correspond to type of open construct:
```

is issued if the type given by the (optional) second word of an END statement does not correspond to the type of the currently open WHEN (4.1), WHENEVER (4.2), EVERY (4.3), or IF (4.4) construct. For an explanation of the correct use of the END statement to close a construct see the indicated section of this User's Guide. The material after the colon is the erroneous word.

### Cuss 231 -- close_name_mismatch

The compile-time error message

> Name in CLOSE statement does not match name of open block:

is issued if the name that may be included as the (optional) third word in a CLOSE statement does not match the name of the innermost block (BUNDLE, SEQUENCE, or SUBSEQUENCE) that is open. The material after the colon is the erroneous word.

### Cuss 232 -- close_incomplete

The compile-time error message

> CLOSE statement must specify type of block being concluded

is issued if the CLOSE statement does not indicate the type (BUNDLE, SEQUENCE, or SUBSEQUENCE) of the block being closed.

### Cuss 241 -- before_within_outside

The compile-time error message

> BEFORE/WITHIN must be inside a WHEN, WHENEVER, or EVERY construct

is issued if a BEFORE or WITHIN statement is encountered that lies outside of any WHEN (4.1.2), WHENEVER (4.2.2), or EVERY (4.3.2) construct. For an explanation of the correct use of the BEFORE and WITHIN statements within such a construct see the indicated section of this User's Guide.

### Cuss 242 -- before_within_already

The compile-time error message

> WHEN/WHENEVER/EVERY construct already has a BEFORE or WITHIN

is issued if a second BEFORE or WITHIN statement is encountered within a WHEN (4.1.2), WHENEVER (4.2.2), or EVERY (4.3.2) construct. For an explanation of the correct use of the BEFORE and WITHIN statements within such a construct see the indicated section of this User's Guide.

**Cuss 243 -- before_within_misplaced**

The compile-time error message

BEFORE or WITHIN must immediately follow WHEN/WHENEVER/EVERY

is issued if a BEFORE or WITHIN statement is encountered that is incorrectly placed within a WHEN (4.1.2), WHENEVER (4.2.2), or EVERY (4.3.2) construct. A BEFORE or WITHIN statement must immediately follow the statement that opens the construct. For an explanation of the correct use of the BEFORE and WITHIN statements within a construct see the indicated section of this User's Guide.

**Cuss 244 -- otherwise_outside**

The compile-time error message

OTHERWISE must be inside a WHEN construct

is issued if an OTHERWISE statement is encountered outside of a WHEN construct. For an explanation of the correct use of the OTHERWISE statement within a WHEN construct see Section 4.1.2 of this User's Guide.

**Cuss 245 -- otherwise_already**

The compile-time error message

WHEN construct already has an OTHERWISE

is issued if a second OTHERWISE statement is encountered inside a WHEN construct. For an explanation of the correct use of the OTHERWISE statement within a WHEN construct see Section 4.1.2 of this User's Guide.

**Cuss 246 -- otherwise_meaningless**

The compile-time error message

OTHERWISE meaningless because there is no BEFORE or WITHIN

is issued if an OTHERWISE statement in encountered inside a WHEN construct that does not contain a BEFORE or WITHIN statement. For an explanation of the correct use of the OTHERWISE statement within a WHEN construct see Section 4.1.2 of this User's Guide.

**Cuss 251 -- else_outside**

The compile-time error message

> ELSE statement must be inside an IF construct

is issued if an ELSE statement is encountered outside of an IF construct. For an explanation of the correct use of the ELSE statement within an IF construct see Section 4.4.

**Cuss 252 -- elseif_outside**

The compile-time error message

> ELSEIF statement must be inside an IF construct

is issued if an ELSEIF statement is encountered outside of an IF construct. For an explanation of the correct use of the ELSEIF statement within an IF construct see Section 4.4.

**Cuss 253 -- else_already**

The compile-time error message

> ELSEIF statement must precede ELSE statement in IF construct

is issued if a second ELSEIF statement is encountered outside an ELSE construct. For an explanation of the correct use of the ELSE IF statement within an IF construct see Section 4.4 of this User's Guide.

**Cuss 261 -- too_many_ss_ops**

The compile-time error message

> Too many CALLs, STARTs, RESUME and STOPs -- maximum number is nssop

is issued if a BUNDLE contains too many combined CALL, START, STOP and RESUME statements. The number of statements of these types that are permitted is given by the parameter "nssop" in the module TL_DATA_COM.

**Cuss 262 -- seq_subseq_not_found**

The compile-time error message

> Referenced SEQUENCE or SUBSEQUENCE is not present -- error in line

is issued if, at the time a given BUNDLE is closed, a SEQUENCE or SUBSEQUENCE called, started, stopped, or resumed within the BUNDLE, is not present in the BUNDLE. The line number specifies the location of the unresolved reference.

**Cuss 271 -- op_requires_seq**

The compile-time error message

START/RESUME/STOP operation not valid for subsequences -- error is in line

is issued if the block referred to by a START, STOP, or RESUME statement is not a SEQUENCE.

**Cuss 272 -- op_requires_subseq**

The compile-time error message

CALL operation not valid for sequences -- error in line

is issued if the block referred to by a CALL statement is not a SUBSEQUENCE.

**Cuss 281 -- too_late_for_defdec**

The compile-time error message

DECLARE or DEFINE must precede first executable statement in block

is issued if a DECLARE or DEFINE statement is encountered, within a BUNDLE, SEQUENCE, or SUBSEQUENCE block, subsequent to the first control or action statement in that block. For proper use of the DECLARE (6.2) and DEFINE (6.1) statement see the referenced section of this User's Guide.

**Cuss 282 -- illegal_defdec_subscript**

The compile-time error message

Reference to declared or defined component cannot be subscripted

is issued if a reference to a component created and named by a DEFINE or DECLARE statement includes a subscript. The ability to subscript a DECLARED internal variable is not supported by the present version of TIMELINER.

**Cuss 291 -- declare_type_missing**

The compile-time error message

> Type of declared internal variable is missing from statement

is issued if a DECLARE statement (Section 6.2) is encountered that lacks an indication of the type (BOOLEAN, NUMERIC, or CHARACTER) of the variable to be created.

**Cuss 292 -- declare_size_no_good**

The compile-time error message

> Size of declared internal variable must be an integer literal

is issued if the size of an internal variable created by a DECLARE statement (Section 6.2) is not expressed as an integer literal.

**Cuss 293 -- declare_size_misplaced**

The compile-time error message

> Size of declared internal variable must follow type

is issued if the size specification in a DECLARE statement (Section 6.2) does not follow the indicator of the variable's type.

**Cuss 301 -- no_as_in_definition**

The compile-time error message

> Definition statement must contain the word 'AS'

is issued if a DEFINE statement (Section 6.1) is encountered that does not contain the word "AS".

**Cuss 302 -- def_not_recognized**

The compile-time error message

> The following definition is not of recognized type:

is issued if the component to the right of the "AS" in a DEFINE statement (Section 6.1) is not of boolean, numeric, character string or range type. The offending item is given after the colon.

**Cuss 311 -- too_many_bool_int_vars**

The compile-time error message

> Too many boolean internal variables declared -- maximum is nbiv

is issued if the number of boolean elements established by declarations of boolean internal variables exceeds a maximum set by the parameter "nbiv" defined in the module TL_DATA_COM.

**Cuss 312 -- too_many_num_int_vars**

The compile-time error message

> Too many numeric internal variables declared -- maximum is nniv

is issued if the number of numeric elements established by declarations of numeric internal variables exceeds a maximum set by the parameter "nniv" defined in the module TL_DATA_COM.

**Cuss 313 -- too_many_char_int_vars**

The compile-time error message

> Too many boolean internal variables declared -- maximum is nciv

is issued if the number of characters established by declarations of character internal variables exceeds a maximum set by the parameter "nciv" defined in the module TL_DATA_COM.

**Cuss 321 -- component_nesting_too_deep**

The compile-time error message

> Component nesting too deep -- maximum number of levels is ncnl

is issued if the depth of component nesting exceeds the capacity of the TIMELINER tables. The number given in the cuss is the maximum number of levels of nesting that can currently be handled. Roughly speaking, an expression in a TIMELINER statement consumes one level of component nesting for each function or operator in the expression. This number is controlled by the parameter "ncnl" in the module TL_DATA_COM. For a given TIMELINER script, the usage of the depth available for component nesting is shown as part of the "file usage summary" printed at the end of the compile-time listing.

**Cuss 331 -- comp_types_disagree**

The compile-time error message

Component types on the two sides of operator do not agree:

is issued if the component types on each side of a comparison operator (i.e. =, /=, <, >, <=, or >=) are incompatible with each other, such that the comparison cannot be evaluated.

**Cuss 332 -- comp_sizes_disagree**

The compile-time error message

Component sizes on the two sides of operator do not agree:

is issued if the components on the two sides of an operator do not have an equal number of elements. This cuss is issued in cases where unequal sizes are not permitted.

**Cuss 333 -- comp_sizes_incompatible**

The compile-time error message

Component sizes on the two sides of operator are not compatible:

is issued if the components on the two sides of an operator do not have compatible numbers of elements. This occurs if the components are both plural, but they are unequal in size.

**Cuss 341 -- component_plural**

The compile-time error message

A plural component is inappropriate in this context:

is issued if a component is arrayed in a case where it must not be. For example, the time interval given in a WAIT statement must be unarrayed. Such a component may be inherently unarrayed, or may be subscripted to indicate a single component.

**Cuss 342 -- component_blank**

The compile-time error message

Parser has isolated a blank component...

is issued if the TIMELINER compiler encounters a case where a component must be present, but no material at all is there.

**Cuss 351 -- comp_not_boolean**

The compile-time error message

> Context requires the following component to be of boolean type:

is issued if a component is not of boolean type in a situation where it must be. For example, this cuss is issued if a numeric or character component is found next to a logical operator such as "AND" or "OR".

**Cuss 352 -- comp_not_numeric**

The compile-time error message

> Context requires the following component to be of numeric type:

is issued if a component is not of numeric type in a situation where it must be. For example, this cuss is issued if a boolean or character component is found next to an arithmetic operator such as "+" or "*".

**Cuss 353 -- comp_not_character**

The compile-time error message

> Context requires the following component to be of character string type:

is issued if a component is not of character string type in a situation where it must be.

**Cuss 354 -- comp_not_recognized**

The compile-time error message

> Parser cannot recognize the following 'thing':

is issued if the "thing" printed after the colon is not a component that can be recognized by TIMELINER.

**Cuss 361 -- stat_needs_boolean_single**

The compile-time error message

> Statement requires singular boolean component -- which the following is not:

is issued if the TIMELINER compiler finds a plural or a non-boolean component in a case where a singular boolean component is required. This cuss is issued of the component following the statement keyword in a WHEN, WHENEVER, BEFORE, IF or ELSEIF statement is other than an unarrayed boolean component.

**Cuss 362 -- stat_needs_numeric_single**

The compile-time error message

Statement requires singular numeric component -- which the following is not:

is issued if the TIMELINER compiler finds a plural or a non-numeric component in a case where a singular numeric component is required. This cuss is issued if the component following the statement keyword in an EVERY, WAIT or WITHIN statement is other than an unarrayed numeric component.

**Cuss 371 -- not_needs_boolean**

The compile-time error message

Component following NOT operator is not of boolean type:

is issued if a numeric or string component is preceded by the "NOT" operator.

**Cuss 372 -- pos_neg_need_numeric**

The compile-time error message

Component following plus or minus operator is not of numeric type:

is issued if a boolean or string component is preceded by a minus or plus sign used as a unary operator.

**Cuss 381 -- num_funct_needs_numeric**

The compile-time error message

Component following numeric funct operator is not of numeric type:

is issued if the component to which a numeric function operator is applied is not of numeric type.

**Cuss 382 -- funct_needs_argument**

The compile-time error message

Function must abut an argument within parentheses:

is issued if a function operator is encountered that is not followed, without intervening spaces, by an argument within parentheses.

**Cuss 291 -- in_needs_range**

The compile-time error message

IN operator requires a range on the right side -- which the following is not

is issued if the component to the right of an "IN" operator is not of range type.

**Cuss 401 -- baffling_operator**

The compile-time error message

The following operator cannot be recognized:

is issued if the parser encounters an operator that it cannot recognize.

**Cuss 402 -- misplaced_operator**

The compile-time error message

The following operator is incorrectly placed:

is issued if an operator is incorrectly placed within a component. For example, this cuss is issued if "NOT" is used an a binary rather than a unary operator.

**Cuss 403 -- missing_operator**

The compile-time error message

Operator is missing from the following expression:

is issued if the printed expression does not have an operator where it must have one. This cuss may result from a blank space accidentally left within a word. This cuss could occur in a case where a statement keyword is misspelled, such that it is appended to the previous statement.

**Cuss 411 -- bad_hanging_operator**

The compile-time error message

The following component improperly ends with an operator:

is issued if an operator is encountered at the end of a component.

**Cuss 412 -- bad_leading_operator**

The compile-time error message

> The following component improperly begins with an operator

is issued if an operator is encountered at the beginning of a component that may not correctly be placed in that position. For example, this cuss is issued if a component begins with a "AND", ">=" or "**".

**Cuss 421 -- parantheses_unbalanced**

The compile-time error message

> The following component contains unbalanced parentheses --
> parsing will fail:

is issued if a component contains an unequal number of open-parentheses and close-parentheses. No further attempt is made to parse the faulty component.

**Cuss 422 -- quotes_unbalanced**

The compile-time error message

> The statement contains unbalanced number of single or double
> quotation marks

is issued if a statement contains an unequal number of single or double quotation marks. However, single quotation marks may be included in a character literal bounded by double quotation marks, or vice versa.

**Cuss 423 -- quotes_in_quotes**

The compile-time error message

> The arrangement of quotation marks is faulty in the following expression:

is issued if a component is encountered bounded by single or double quotation marks, that contains two or more quotation marks of the same type.

**Cuss 431 -- subscript_is_var_range**

The compile-time error message

> A variable range may not be used in a subscript -- use fixed range:

is issued if the subscript field of an array is specified as a variable range, i.e. a range one or both of whose bounds are specified by a component that is not a numeric literal.

**Cuss 432 -- subscript_is_scal_lit**

The compile-time error message

> A scalar literal may not be used in a subscript -- use integer literal:

is issued if the subscript field of an array is specified as a scalar literal, i.e. a numeric literal containing a decimal point.

**Cuss 441 -- subscript_is_not_numeric**

The compile-time error message

> A non-numeric component may not be used in a subscript:

is issued if a component is encountered as part of a subscript field that is not of numeric or range type.

**Cuss 442 -- subscript_is_plural**

The compile-time error message

> A plural component may not be used in a subscript:

is issued if a component found within a subscript has more than one element. Arrayed components cannot be used as subscripts.

**Cuss 451 -- subscript_field_bogus**

The compile-time error message

> The following subscript field is improperly formatted:

is issued if there is a formatting error within a variable's subscript.

**Cuss 452 -- subscript_out_of_range**

The compile-time error message

> The following subscript is out-of-range for this variable:

is issued if a variable's subscript is specified by a literal that is outside of the range that is legal for the variable.

## Cuss 453 -- too_few_subscripts

The compile-time error message

```
Too few subscripts -- this variable requires
```

is issued if an arrayed simulation variable used in a LOAD or PRINT statement, or as a component in any statement, has been entered with too few subscripts. If an array is subscripted at all, the number of subscripts given must correspond to the number of dimensions in the array.

**Cuss 454 -- too_many_subscripts**

The compile-time error message

> Too many subscripts -- this variable requires

is issued if a variable used in a LOAD or PRINT statement, or as a component in any statement, has been entered with too many subscripts. If an array is subscripted at all, the number of subscripts given must correspond to the number of dimensions in the array. This cuss is issued if an unarrayed variable is subscripted at all.

**Cuss 461 -- fixed_range_out_of_order**

The compile-time error message

> Left range parameter must be less than right range parameter:

is issued if a range component is encountered, in which both ends of the range are expressed as a numeric literal and the value given for the start of the range exceeds the value given for the end of the range.

**Cuss 462 -- comp_not_variable**

The compile-time error message

> Context requires the following component to be a variable:

is issued if the encountered component is not a variable. This cuss is issued if the component to the left of the assignment operator in a LOAD statement is not a simulation variable or a declared internal variable.

**Cuss 471 -- load_delimiter_missing**

The compile-time error message

> LOAD statement does not contain an assignment delimiter:

is issued if a LOAD statement is encountered that does not contain a delimiter such as "=" or ":=" between the variable and the data to be loaded into the variable.

**Cuss 472 -- load_data_missing**

The compile-time error message

> LOAD statement contains no data to be loaded

is issued if a LOAD statement is encountered that contains no data to be loaded into the variable.

**Cuss 473 -- load_sizes_incompatible**

The compile-time error message

> LOAD data must be singular or match size of variable:

is issued if a LOAD statement is encountered in which the component to the right of the assignment operator is plural, but has a number of elements that is either greater or smaller than the number required to LOAD the component to the left of the operator. If the size of the right-hand component is singular the same value will be loaded into every element of a plural component to the left.

**Cuss 481 -- load_data_not_boolean**

The compile-time error message

> LOAD variable is boolean, but the data to be loaded is not:

is issued if the encountered LOAD statement specifies the loading of boolean data into a non-boolean variable.

**Cuss 482 -- load_data_not_numeric**

The compile-time error message

> LOAD variable is numeric, but the data to be loaded is not:

is issued if the encountered LOAD statement specifies the loading of numeric data into a non-numeric variable.

**Cuss 483 -- load_data_not_cstring**

The compile-time error message

> LOAD variable is character string, but the data to be loaded is not:

is issued if the encountered LOAD statement specifies the loading of character string data into a non-character string variable.

**Cuss 491 -- cannot_load_event**

The compile-time error message

> The following is an event, so please use SIGNAL or CLEAR instead of LOAD:

is issued if the encountered LOAD statement specifies the loading of an EVENT. The SIGNAL (5.4) and CLEAR (5.5) statements must be used to set or reset an event.

### Cuss 492 -- set_reset_require_event

The compile-time error message

> SIGNAL and CLEAR statements require an event, which the following is not

is issued if a SIGNAL or CLEAR statement is encountered in which the statement keyword is not followed by a component representing a simulation event known to TIMELINER.

### Cuss 501 -- mess_data_not_character

The compile-time error message

> Message statement requires character string, which this ain't:

is issued if the statement keyword in a MESSAGE statement (Section 5.9) is not followed by a component of character string type.

### Cuss 511 -- negative_time_interval

The compile-time error message

> Literal time interval in EVERY, WITHIN or WAIT statement is negative:

is issued if the time interval specified in an EVERY, WITHIN, or WAIT statement is a negative numeric literal. If the time interval is given in the form of a variable no cuss can be issued, of course, because the value of the variable is unknown at compile time. A negative time interval discovered at run-time will result in a zero interval.

## 9.0 TIMELINER MAINTENANCE

Under the heading "TIMELINER maintenance" are included those functions that may occasionally be necessary to adapt the version of TIMELINER dedicated to a particular application to the needs of that application.

This chapter has two parts. The first discusses how to change the sizes of the tables used by TIMELINER to store executable code. The second discusses how to make TIMELINER aware of the variables that form part of the application.

## 9.1 How to Change the Size of the TIMELINER Tables

TIMELINER works by analyzing the raw script input by the user and placing the information contained in the script in a set of tables. The tabulated script information for each bundle (Section 2.1) is then written into a file. At execution-time, the information is read from the file into an identical set of tables, where it is executed.

The size of the TIMELINER tables is determined by a series of parameters. The following table states the name of the parameter, the Ada module where the parameter is defined, and the function of each parameter:

| | | |
|---|---|---|
| nll | TL_DATA_COM | max length of raw line read from file |
| nls | TL_DATA_COM | max length of statements |
| nwl | TL_DATA_COM | word length |
| nbk | TL_DATA_COM | number of bundles, sequences, and subsequences |
| nnm | TL_DATA_COM | number of names |
| nst | TL_DATA_COM | number of statements |
| ncd | TL_DATA_COM | amount of component data |
| nnl | TL_DATA_COM | number of numeric literals |
| nsl | TL_DATA_COM | number of string literals |
| nbb | TL_DATA_COM | size of boolean buffers |
| nnb | TL_DATA_COM | size of numeric buffers |
| nsb | TL_DATA_COM | size of string buffers |
| neb | TL_DATA_COM | size of event buffers |
| nsub | TL_DATA_COM | number of subscripts that a variable may have |
| nbnl | TL_DATA_COM | maximum block nesting level |
| nsnl | TL_DATA_COM | maximum statement nesting level |

| | | |
|---|---|---|
| ncnl | TL_DATA_COM | maximum component nesting level |
| ndd | TL_DATA_COM | maximum number of definitions/declarations |
| npl | TL_DATA_COM | maximum column length of TIMELINER printing |
| nssop | TL_INIT_COM | maximum number of seq/subseq operations |

Each BUNDLE's usage of the tables sized by these parameters is printed as part of the "file usage summary" that forms part of the compile-time listing of a TIMELINER script.

A particular embodiment of the Ada-language version of TIMELINER may require varying amounts of storage, depending upon the length and level of complexity of the scripts that are required. Therefore it is anticipated that a particular TIMELINER application will need to tailor the table sizes controlled by the parameters listed above to meet its particular needs.

## 9.2 How to Make TIMELINER Aware of Simulation Variables

At the core of TIMELINER lies a complex set of logic that makes TIMELINER aware of the variables that form part of the application to which a version of TIMELINER is attached.

Variables of boolean type, numeric type, character string type, and event type may be manipulated by TIMELINER. Variables may be arrayed in up to three dimensions.

The TIMELINER logic that provides access to variables is referred to as the "low-level" TIMELINER logic. The changes required to adapt TIMELINER to a particular application are confined to this low-level logic. TIMELINER's "upper-level" logic, which implements the language features, is the same for all applications.

The TIMELINER low-level logic is accessed by means of three separate functions:

| | |
|---|---|
| **FIND_VAR** | Used at compile-time only. Returns an index by which the variable can be referenced during execution. |
| **GRAB_VAR** | Obtains the value of a variable (which may be arrayed) for use by a control statement in making a decision. |
| **LOAD_VAR** | Loads the variable (which may be arrayed) using data supplied by the user. |
| **PRINT_VAR** | Prints (or otherwise displays) the value of a variable (which may be arrayed). |

In order to perform these functions with respect to a particular variable, TIMELINER requires that the variable be entered into the

TIMELINER "variable list" found in the Ada package TL_VAR_LIST. This process consists of three steps:

- The package specification in which the variable resides must be known to the TIMELINER compilation unit. This is accomplished by 'with'ing the package specification at the top of the compilation unit that contains the variable list.

- The variable must be entered alphabetically into the variable list, including the attributes described below.

- The variable LIST_SIZE must be adjusted to reflect the actual number of variables in the variable list.

The variable attributes for each variable contained in the variable list are:

**VAR_NAME_STRING** A character string representing the name of the variable in question. The variable list must be structured in alphabetical order, according to VAR_NAME_STRING. In the present version of TIMELINER, VAR_NAME_STRING is a string of 40 characters. If the variable appears in more than one package specification, the variable name should be prefaced by the package name.

**VAR_VAL_TYPE** The variable type. The types that the present version of TIMELINER is equipped to handle are specified by the following names:

| | |
|---|---|
| OF_SCALAR_SINGLE | 4-byte float with 6-digit precision |
| OF_SCALAR_DOUBLE | 8-byte float with 12-digit precision |
| OF_INT8 | 1-byte integer |
| OF_INT16 | 2-byte integer |
| OF_INT32 | 4-byte integer |
| OF_BOOLEAN | 1-byte boolean |
| OF_BOOL32 | 4-byte boolean |
| OF_CHARACTER | 1-byte character |
| OF_EVENT | CIFO event |

New types may be defined by the user in package TL_VAR_LIST as explained below.

**TLINER_VAL_TYPE** The TIMELINER type that is most appropriate for the variable in question. The choices are NUMBER_TYPE, BOOLEAN_TYPE, CHARACTER_TYPE, and EVENT_TYPE, corresponding to the components that are described in Section 7.4 of this document.

**NUMB_DIMENSIONS** The number of dimensions associated with a variable. This number must lie between 0 (un-arrayed variable), and 3 (three-dimensional array). In the case of character variables one of the dimensions establishes the string length, and the other two determine how strings of the specified length are arrayed.

**MAX_SUB_1** The size of the first dimension of a variable arrayed in one, two, or three dimensions. The variable's first subscript must fall into the range between 1 and this number.

**MAX_SUB_2** The size of the second dimension of a variable arrayed in two, or three dimensions. The variable's second subscript must fall into the range between 1 and this number.

**MAX_SUB_3** The size of the third dimension of a variable arrayed in three dimensions. The variable's third subscript must fall into the range between 1 and this number.

**VAR_SYS_ADDR** The address, in a particular processor, at which the variable resides. If the variable appears in more than one package specification, this address entry must be prefaced by the package name.

*In the single-processor (functional) version of the CSDL real-time SSF DMS testbed, the variable list is located in the package TL_VAR_LIST found in the file TL_VAR_LIST_SP_S.ADA.*

When TIMELINER is incorporated into a multi-processor architecture, a separate variable list must be maintained for each processor forming part of the system. In this case the following attribute must be added to each entry in a variable list:

**PROCESSOR** An identifier that names the processor in which the variable resides. The user will define, as explained below, the valid choices for this identifier. *(In the case of the CSDL real-time testbed the available processor identifiers are ENV_PROC and FSW_PROC.)*

In a multi-processor environment, the variable list for each processor is sized by a separate parameter. Each list size parameter must correspond to the number of variables in each respective list.

Here is an example of a variable list entry for variable ATMOS_MODEL, which in this case is a 2x2x2 array of 32-bit integer type:

```
("ATMOS_MODEL                        ", -- Variable name
 OF_INT32,                              -- Variable value type
 NUMBER_TYPE,                           -- TIMELINER type
 3,                                     -- Number of dimensions
 2,                                     -- Size of first dimension
 2,                                     -- Size of second dimension
 2,                                     -- Size of third dimension
 ATMOS_MODEL'ADDRESS,                           -- Variable address
 ENV_PROC),                                     -- Processor where variable resides
```

For use during execution time, the information in the individual variable lists is transferred to a composite variable list called REMOTE_VAR_LIST that lies within the processor where TIMELINER operates. The parameter that sizes the composite list must be equal to the sum of the sizes of the individual lists resident in each processor.

Variables entered in the variable lists must be in alphabetical order. The compiler directive CHECK_VAR_LIST, described in section 2.7, is provided to ensure that the variables are entered in the proper order. It also checks for errors in the dimensionality of a variable.

*In the multiple-processor version of the real-time testbed, the variable list ENV_LIST for the "environment" processor is located in the file ENV_PROC_LIST_S.ADA and the variable list FSW_LIST for the "flight software" processor is located in the file FSW_PROC_LIST_S.ADA. The composite list REMOTE_VAR_LIST is located in the package TL_VAR_LIST found in the file TL_VAROPS_LIST_MP_S.ADA. In this case the composite list size LIST_SIZE must equal the sum of ENV_LIST_SIZE and FSW_LIST_SIZE.*

As stated above, the user may wish to add new variable types to the ones already available, as listed above under the variable list attribute VAR_VAL_TYPE. Such additional types may be defined by the user in package TL_VAR_LIST.

For example, if an 8-bit boolean is required by the implementation, it would be defined by the statements

```
      type BOOL8 is new BOOLEAN;
      for BOOL8'SIZE use 8;
```

Note that Appendix F of the appropriate Compiler User's Guide should be consulted to find the representation of the various base types available to the user for the particular compiler being used.

To incorporate the new type into TIMELINER, the enumeration type VARIABLE_VALUE_TYPE must be modified to include the new type, for example:

```
      type VARIABLE_VALUE_TYPE is  (OF_SCALAR_SINGLE,
                                    OF_SCALAR_DOUBLE,
                                    OF_INT8,
                                    OF_INT16,
                                    OF_INT32,
                                    OF_BOOLEAN
                                    OF_BOOL8,
                                    OF_BOOL32,
                                    OF_CHARACTER,
                                    OF_EVENT);
```

The number of 8-bit bytes allocated for storage of the newly defined variable type is then represented in NUMB_STORAGE_BYTES, and the number of characters needed to print each element is represented in PRINT_SPACE as follows:

```
NUMB_STORAGE_BYTES : VAR_TYPE_SIZE := (OF_SCALAR_SINGLE => SCALAR_SINGLE'SIZE/8,
                                       OF_SCALAR_DOUBLE => SCALAR_DOUBLE'SIZE/8,
                                       OF_INT8          => INT8'SIZE/8,
                                       OF_INT16         => INT16'SIZE/8,
                                       OF_INT32         => INT32'SIZE/8,
                                       OF_BOOLEAN       => BOOLEAN/SIZE/8,
                                       OF_BOOL8         => BOOL8'SIZE/8,
                                       OF_BOOL32        => BOOL32'SIZE/8,
                                       OF_CHARACTER     => CHARACTER'SIZE/8,
                                       OF_EVENT         => EVENT'SIZE/8);

PRINT_SPACE : VAR_TYPE_PRINT_SPACE := (OF_SCALAR_SINGLE => 15,
                                       OF_SCALAR_DOUBLE => 21,
                                       OF_INT8          => 7,
                                       OF_INT16         => 9,
                                       OF_INT32         => 14,
                                       OF_BOOLEAN       => 8,
                                       OF_BOOL8         => 8,
                                       OF_BOOL32        => 8,
                                       OF_CHARACTER     => 1,
                                       OF_EVENT         => 7)
```

**WARNING:** TIMELINER is very dependent on the specific types defined by the enumeration type VARIABLE_VALUE_TYPE and further described in NUMB_STORAGE_BYTES and PRINT_SPACE. If new enumeration literals are added to this list or if the names of the enumeration literals are changed, the code in package TL_VAR_OPERATIONS must be modified. Specifically, the control structures that are dependent on these enumeration literals must be changed to reflect the newly defined types.

*In the CSDL real-time testbed the package TL_VAR_OPERATIONS is found in the file TL_VAROPS_SP_B.ADA in the case of the single-processor version, and in the file TL_VAROPS_MP_B.ADA in the case of the multi-processor version.*

For the multi-processor environment, as explained above, the variable attribute **PROCESSOR** identifies the processor in which the variable resides. The processor identifiers are defined by the enumeration type PROCESSOR_TYPE, as illustrated in the following example:

```
type PROCESSOR_TYPE is (ENV_PROC, FSW_PROC);
```

New processor types may be defined by the user by modifying PROCESSOR_TYPE in package TL_VAR_LIST.

**WARNING:** TIMELINER is very dependent on the literals defined by the enumeration type PROCESSOR_TYPE. If new enumeration literals are added to this list, or if the names of the enumeration literals are changed, a major update of the code in package TL_VAR_OPERATIONS is required. Specifically, the control structures that are dependent on these enumeration literals must be changed to reflect the newly defined literals.

Further information about how TIMELINER uses the variable lists to "grab", "load", and "print" variables is given below in Appendix C.

# AN INTRODUCTION TO TIMELINER

February, 1992

Don Eyles
(617) 258-2460

The Charles Stark Draper Laboratory, Inc.
Cambridge, MA 02139

DRAPER
LABORATORY

# Problem 1 -- Simulation

- Need to
  - Initialize
  - Inject errors, perturbations
  - Simulate human inputs
  - Run a series of cases
  - Recover *ad hoc* information
  - Avoid recompilations, relinks

# Problem 2 -- "Flight" System

- Need to
  - Lower workload of human operator
  - Automate procedures, including
    - Spacecraft operation
    - Payload operations
    - Failure recovery
  - Provide upper-level control during LOS when unmanned
  - Allow procedures to be defined "pre-flight" to aid verification, insure repeatability

# Solution

- A language specialized for the writing of sequencing procedures
  - Time-oriented
  - Conditional logic
  - Parallel activities
  - Controllable by user
  - English-like to facilitate writing and monitoring of scripts
- i.e. TIMELINER

# TIMELINER History

2/92

- Apollo ancestry -- "astronaut" sim
- HAL version, about 1981
  - Used to control space shuttle simulations at CSDL
  - Scripts currently in use have over 2000 statements organized into over 100 sequences
- Fortran version, about 1986
  - Used by AFE Fortran simulation
- Ada version in 1990 for SSF real-time testbed
  - Functional and real-time multi-processor versions
  - Used by test-bed and Mars autonomous lander sims
- Selected in 1991 for use aboard space station

new

# Current Users

- OFS20 -- HAL
  - Shuttle DAP (Neil Adams, Daryl Sargent)
  - Shuttle On-orbit Integrated GN&C (Peter Kachmar)
  - Shuttle Entry Nav
  - Dual (two spacecraft) OFS (Marty Matusky)

- Hughes -- HAL (Peter Weiler)

- AFE -- HAL, Fortran, Ada (Ken Spratlin)

- Mars autonomous lander -- Ada, HAL (Tony Bogner, Ken Spratlin)

- SSF test-bed -- Ada (Roger Racine)
  - Functional and real-time multi-processor

# Design Concept

- **TIMELINER design based on paradigms:**

Ⓐ

```
WHEN  condition
   action
   action
WAIT  time_interval
   action
   action
WHEN  condition
   action
   action
 etc....
```

Ⓑ

```
WHENEVER  condition
   action
   action
```

Ⓒ

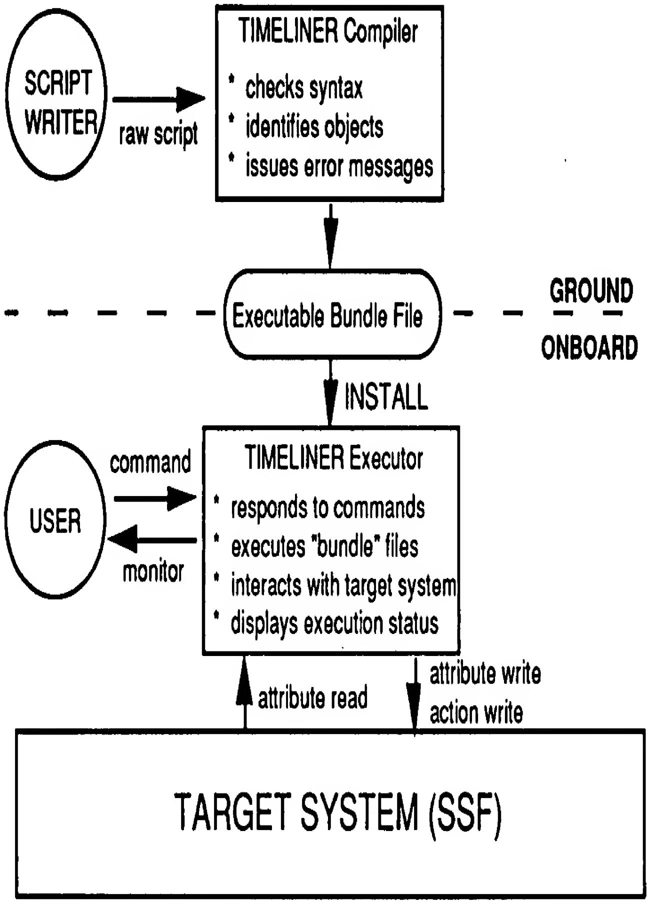
```
EVERY  time_interval
   action
   action
```

- **Multiple sequences, of whatever type, must act in parallel with each other**

- **"Condition" should be general**

# Operational Context

- Timeliner scripts are compiled and "downloaded" to onboard executor — similar to Apollo EMPs



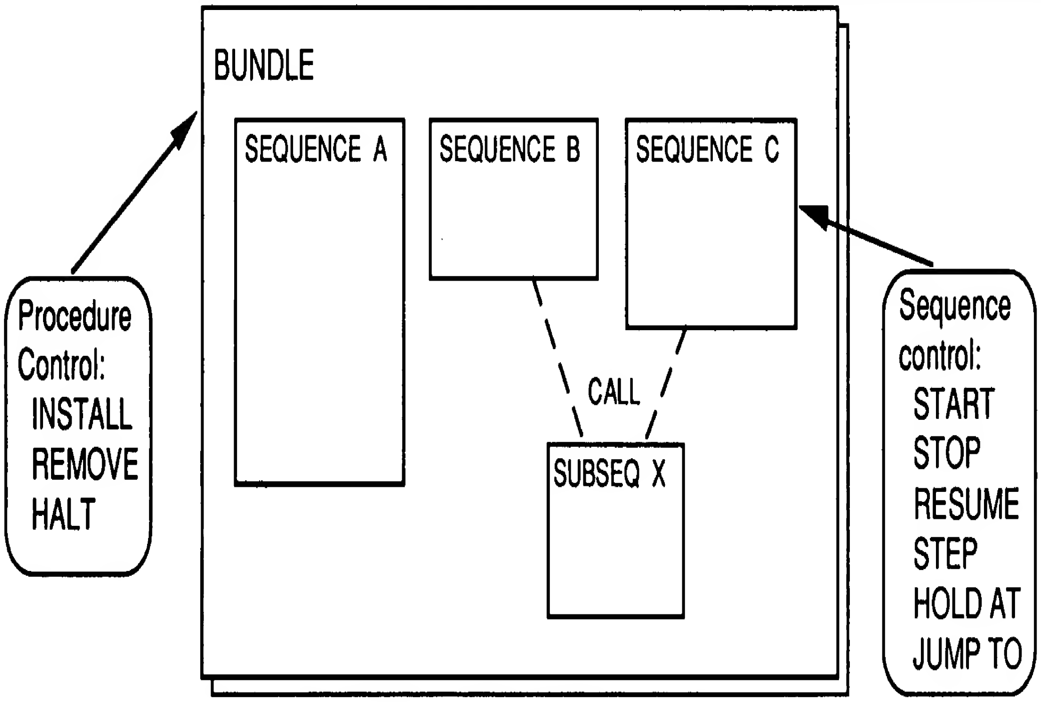
- Executor accepts real-time commands

- Interface with "target" system varies according to the application

new

# Script Organization

2/92



BUNDLE

SEQUENCE A

SEQUENCE B

SEQUENCE C

CALL

SUBSEQ X

Procedure Control:
INSTALL
REMOVE
HALT

Sequence control:
START
STOP
RESUME
STEP
HOLD AT
JUMP TO

- **Compiled bundles can be installed on command**
- **Each bundle is made up of sequences that execute in parallel with each other**
- **Sequences can be started, stopped (etc.) individually**
- **Sequences can CALL subsequences**

# Statement Types

- TIMELINER language includes statements of four types:
  - Blocking statements -- mark the beginning and end of BUNDLEs, SEQUENCEs, and SUBSEQUENCEs
  - Control statements -- used to determine the conditions under which actions will occur
  - Action statements -- used to carry out actions with respect to the target system
  - Non-executable statements
- Statements are made up of keywords and "components"
- All statements begin with a keyword that indicates and determines the statement type

# Blocking Statement Types

- BUNDLE, SEQUENCE, and SUBSEQUENCE headers
- CLOSE statement
- Example:

```
BUNDLE  KU_BAND_OPS

    SEQUENCE  INITIALIZE
       <statements>
    CLOSE SEQUENCE

    SEQUENCE  ANTENNA_DEPLOY  INACTIVE
       <statements>
    CLOSE SEQUENCE

    SUBSEQUENCE  POWER_ON
       <statements>
    CLOSE SUBSEQUENCE

CLOSE BUNDLE
```

new

# Control Statement Types

2/92

- WHEN -- wait for some condition, <u>once</u>
- WHENEVER -- wait for some condition, <u>repeatedly</u>
- EVERY -- create loop at some interval
- BEFORE, WITHIN, OTHERWISE -- used to modify WHEN, WHENEVER, EVERY
- IF, ELSEIF, ELSE -- non-time-oriented conditional
- END -- closes WHEN, WHENEVER, EVERY, IF construct
- WAIT -- simple pause
- Examples on next slide...

new

# Control Statement Examples

- Simplest form of WHEN construct...

```
WHEN GMT >= 10:15:33 CONTINUE
```

- Do different actions depending on what condition occurs first

```
WHEN ALTITUDE < 400000
    BEFORE EVENT_301
       SET AERO_MODEL TO 1
    OTHERWISE
       SET AERO_MODEL TO 0
END WHEN
```

- Do something every time that a condition occurs...

```
WHENEVER TEMP > 80 and STATUS of HEATER = ON
    COMMAND HEATER TURN_OFF
END WHENEVER
```

- Loop at some interval until some condition occurs

```
EVERY 2.0
    WITHIN 300
        PRINT CABIN_TEMP
END EVERY
```

# Action Statement Types

2/92

- START, STOP, RESUME -- control other sequences

- Other action statements depend upon application, e.g.

  - Simulation application
    - LOAD, PRINT, DUMP variables
    - EXECUTE arbitary software modules
    - Provide human inputs such as keystrokes
    - Control system operation

  - SSF onboard application
    - Ability to command actions (including actions to control TIMELINER execution) using DMS services
    - Ability to write attributes using DMS services

# Non-Executable Statement Types

- Scope is entire BUNDLE, or entire SEQUENCE or SUBSEQUENCE -- must be located at top of block

- DECLARE -- create a local variable, e.g.

```
DECLARE TARG_VECT NUMERIC(3)
```

- DEFINE -- apply a name to a component, e.g.

```
DEFINE TEMP_RANGE AS 100 TO 125

WHENEVER TEMP OUTSIDE TEMP_RANGE
    CALL ADJUST_TEMP
END WHENEVER
```

# TIMELINER "Objects"

2/92

- Objects: variables (subscripted), literals, combinations, functions, lists, definitions, keystrokes, ranges

  - Only three "types": boolean, numeric, character string

  - A "condition" may be any unarrayed boolean object

  - Combinations: OR AND = /= < <= > >= IN OUTSIDE .. + - * / MOD ** &

  - Language provides a few simple functions, e.g. ABS

  - Definitions: apply a name to an object, e.g.

```
DEFINE TEMP_RANGE AS 100 TO 125
WHENEVER TEMP OUTSIDE TEMP_RANGE
   CALL ADJUST _TEMP
END WHENEVER
```

  - For SSF add interface with RODB objects / attributes / actions

# TIMELINER Components

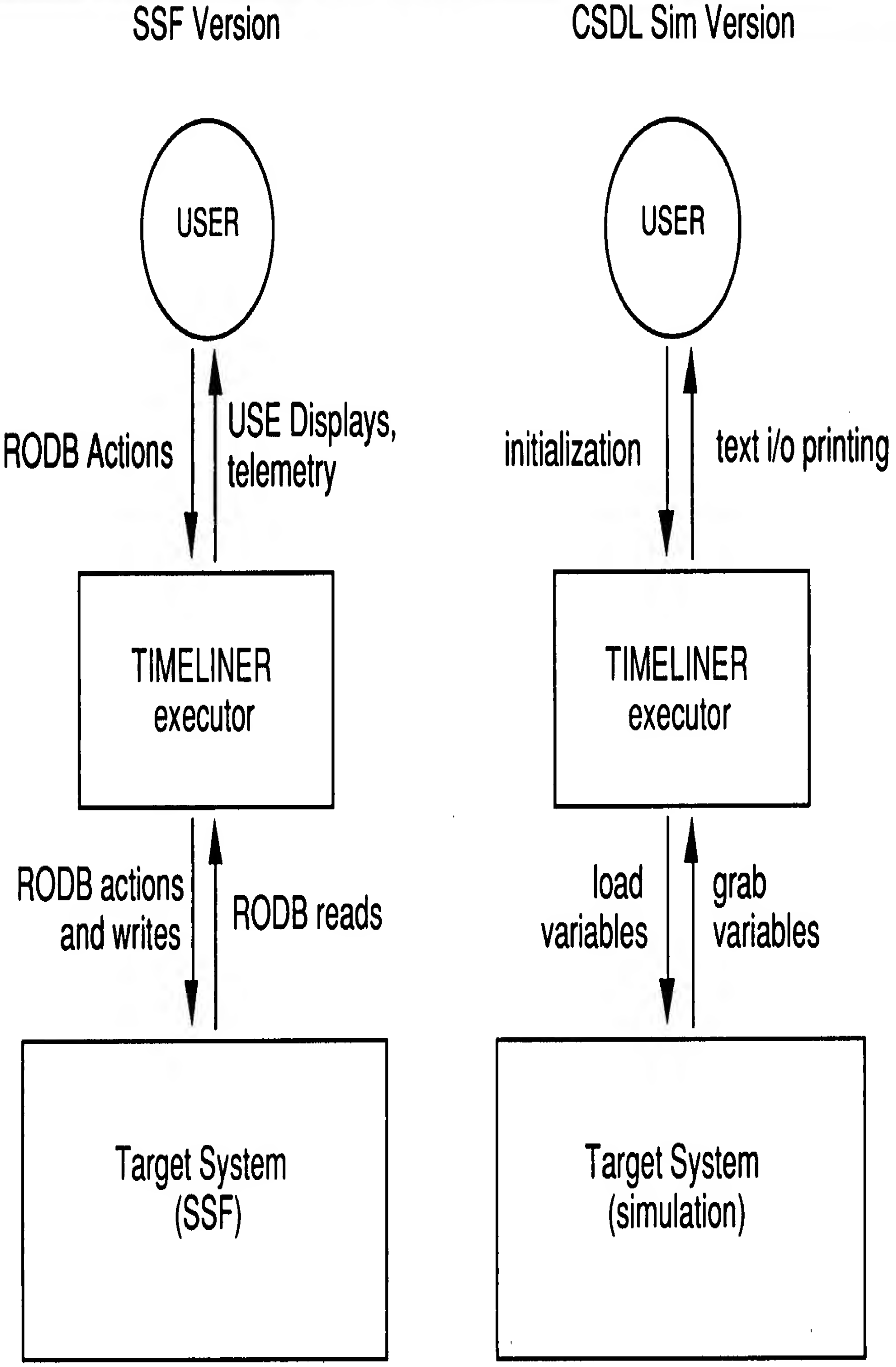
- Components: target system variables (sim version), object attributes and actions (SSF version), combinations, functions, constants, lists, definitions, declared internals, ranges

  - Only three "types": boolean, numeric, character string

  - Combinations: OR AND = /= < <= > >= IN OUTSIDE TO + - * / MOD ** ||

  - Language provides a few simple functions, e.g. ABS, SIN, COS, TAN, and conversion constants, e.g. RAD_TO_DEG

# Flavors of TIMELINER



DRAPER LABORATORY

new

# User-Control of TIMELINER Operation

- User-control embodied in actions on object class "UIL Executor"

- Action applying to whole executor
  - FREEZE_ALL -- panic stop all sequences in all installed bundles

- Actions applying to a "bundle"
  - INSTALL -- get bundle from storage and prepare for execution
  - REMOVE -- discards bundle -- all sequences must be stopped
  - HALT -- stop all sequences in bundle

- Actions applying to "sequence"
  - START -- starts sequence at first statement
  - STOP -- unconditional stop, statement pointer not reset
  - PROCEED -- starts sequence from current point
  - STEP -- executes one statement from current point
  - HOLD_AT -- sets one-time breakpoint at specific statement
  - JUMP_TO -- jumps to specific statement, legal when sequence is stopped and to/from statements are not inside any construct

# Display Interface

- Display requirements / design should come from user
- USE display capabilities not well defined yet
- Our approach has been to define attributes that include complete status information
- MOD needs
  - Name of sequence
  - Time at which sequence started
  - Number of current statement
  - Does not need scrolling script display -- but would like it
- Suggested sample displays will be presented in separate pitch

# Interface with DMS

- Interface with RODB via DMS standard services
  - TIMELINER's needs are similar to display's
  - TIMELINER operates as an application package using standard APID interfaces -- no custom services required
  - Each "bundle" has its own executor task
    - Set up cyclic read of objects / attributes at INSTALL time
  - Syntax for dealing with RODB objects / attributes / actions
    - Attribute read: <attribute> OF <object>
      - e.g. WHENEVER SPEED OF PUMP1 > 100
    - Attribute write: SET <attribute> OF <object> TO <value>
    - Action command: COMMAND <object> <action> [<params>]
      - e.g. COMMAND UMBILICAL EXTEND

# MODBM Compile-Time Interface

2/92

- TIMELINER statements reference RODB entities by name

- Compiler must obtain integer IDs corresponding to objects, attributes, actions, and action parameters -- also need attribute and parameter types

- This information is available in MODB

- Three options for obtaining identifiers
  - Direct entry of IDs by writer of script -- plus DEFINEs
  - Use of existing look-up tool -- preferred alternative
  - Creation of look-up tool using Ada packages from MODBM

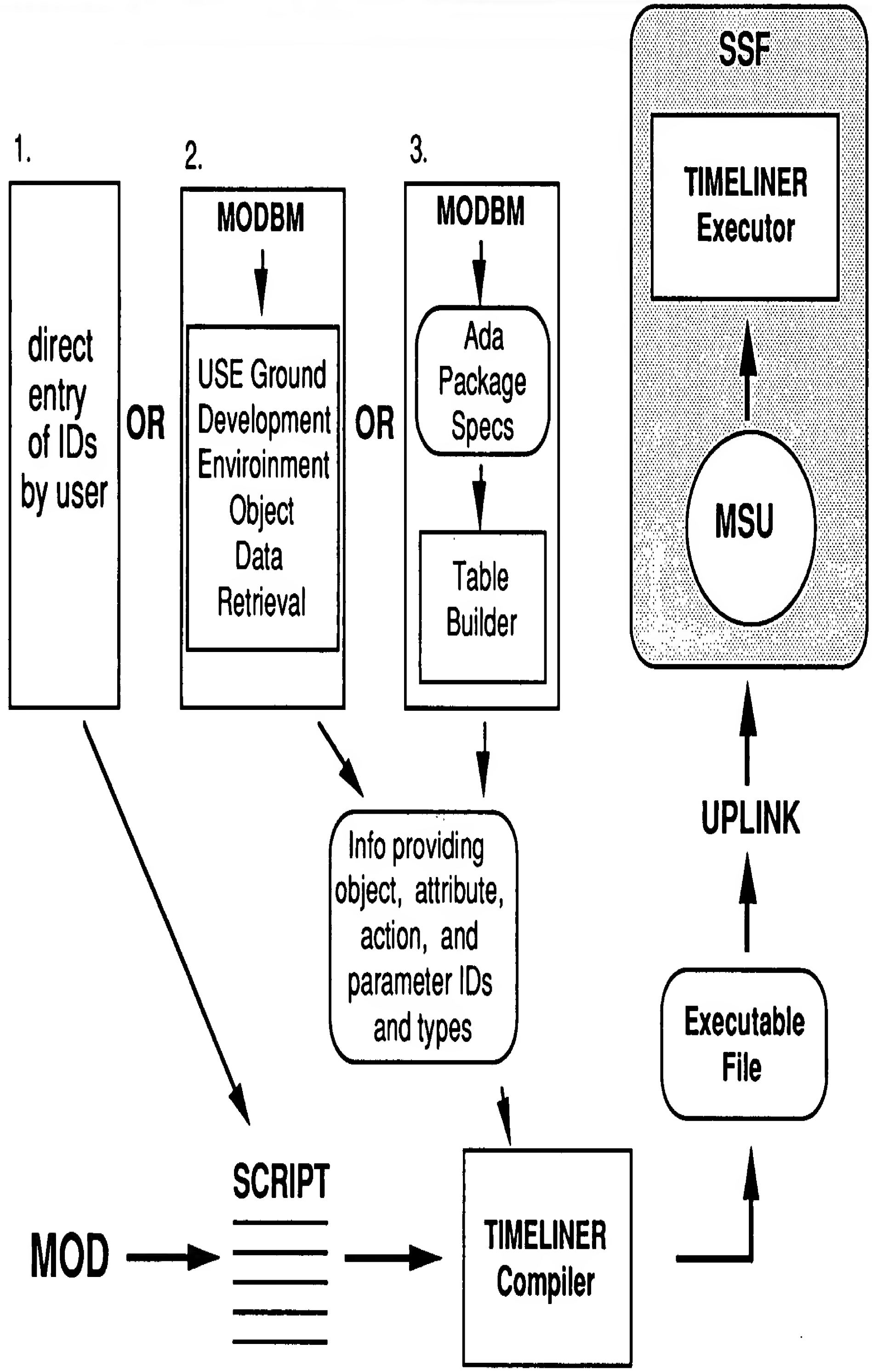
- It's an issue of access to the information, not technical risk

DRAPER LABORATORY

new

# MODBM Compile-Time Interface -- Illustrated

2/92


SSF
TIMELINER Executor
MSU
1.
direct entry of IDs by user
OR
2.
MODBM
USE Ground Development Enviroinment Object Data Retrieval
OR
3.
MODBM
Ada Package Specs
Table Builder
Info providing object, attribute, action, and parameter IDs and types
UPLINK
Executable File
SCRIPT
MOD
TIMELINER Compiler

new

# Interface with MOD, crew, ISE

2/92

- A "bundle" is a group of sequences and subsequences that implement a given ISE activity
  - Each bundle will include "master" sequence to start other sequences in same bundle, on triggering event
  - For a given activity, event handling can be packaged in same bundle as sequencing functions

- Bundles reside in MSU as files of data

- Provide "INSTALL <bundle>" capability callable by ISE
  - Read bundle from mass memory into internal buffers
  - Schedule executor task for bundle using DT and PRIO assigned by Tier I
  - Set up I/O using standard services

- Provide "REMOVE <bundle>" capability callable by ISE
  - Clean up buffers, cancel executor task

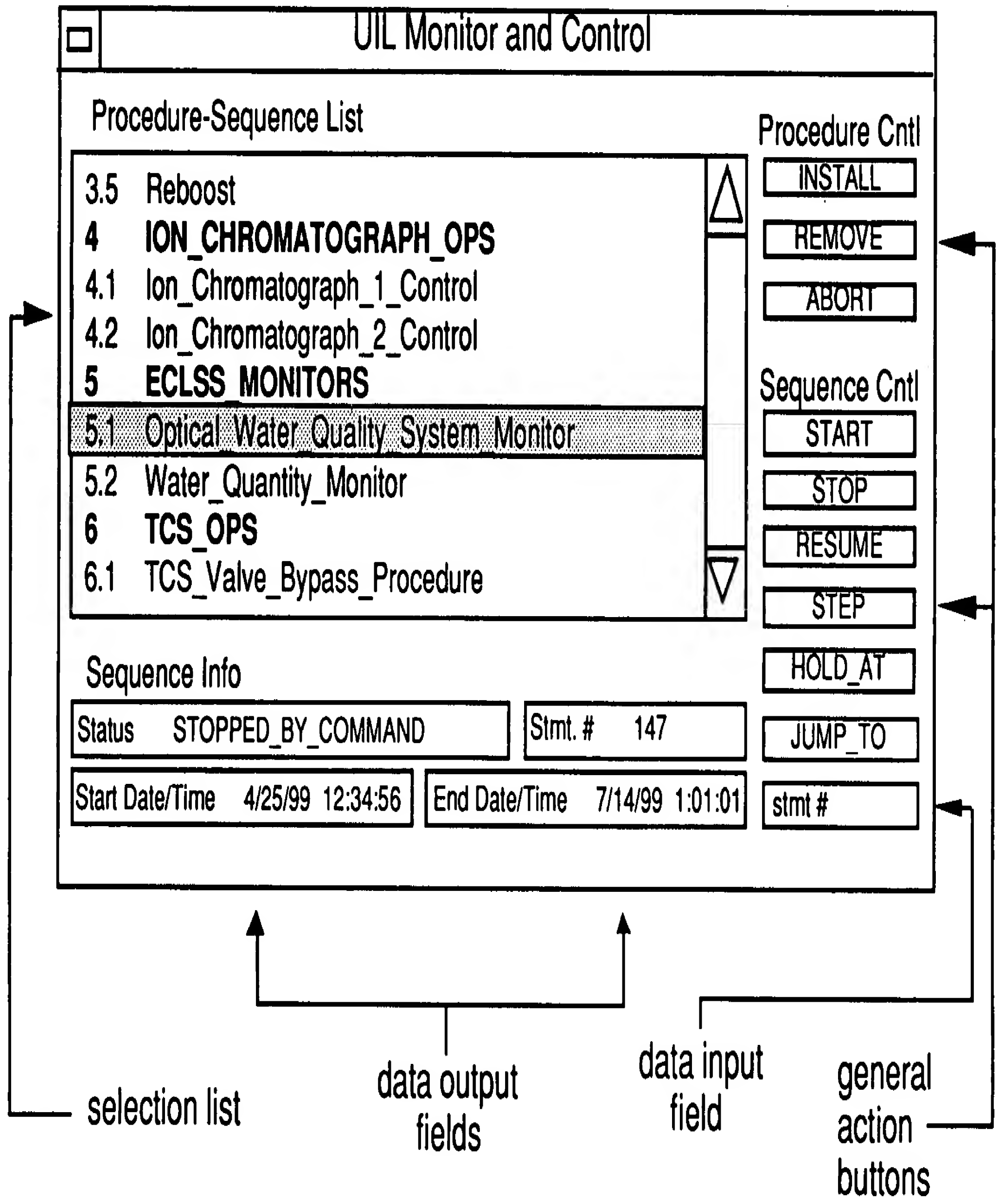
# Interface with Display

- Interface with displays for monitoring
  - Use existing capability to reconstruct source statements from executable code, i.e. de-compile — OR, carry onboard a human-readable copy of each bundle
  - Support menu displays showing status of all procedures, and sequences
  - Support ability to monitor selected sequence -- via scrolling display
  - Indicate current statement by color or graphic
  - Optional single-step operation
  - Possible tutorial or "dialog box" interface

new

# UIL Monitor and Control Display

2/92



UIL Monitor and Control

Procedure-Sequence List

3.5 Reboost
**4 ION_CHROMATOGRAPH_OPS**
4.1 Ion_Chromatograph_1_Control
4.2 Ion_Chromatograph_2_Control
**5 ECLSS_MONITORS**
5.1 Optical_Water_Quality_System_Monitor
5.2 Water_Quantity_Monitor
**6 TCS_OPS**
6.1 TCS_Valve_Bypass_Procedure

Sequence Info

Status STOPPED_BY_COMMAND

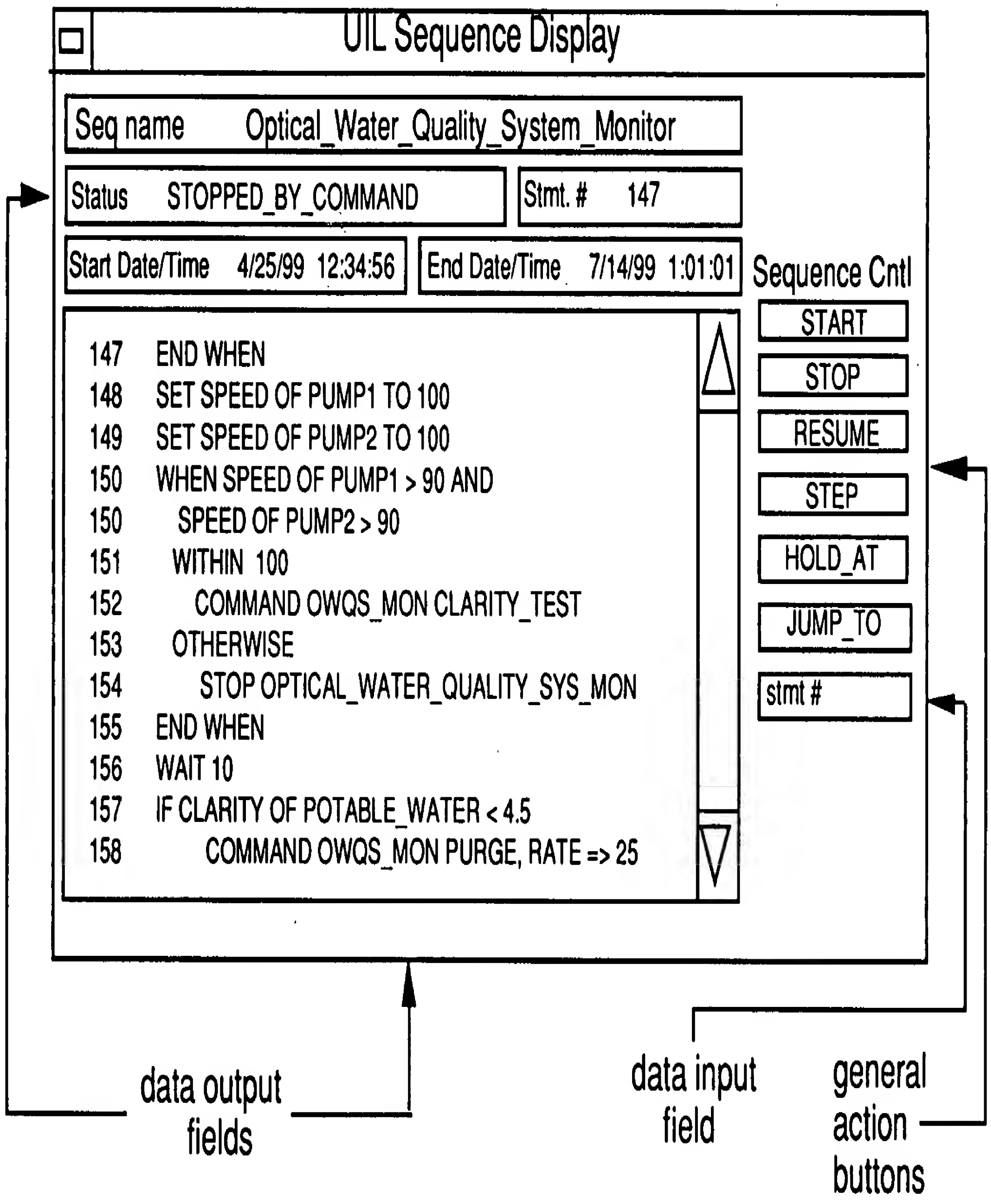
Stmt. # 147

Start Date/Time 4/25/99 12:34:56

End Date/Time 7/14/99 1:01:01

# UIL Sequence Display

2/92



UIL Sequence Display

Seq name Optical_Water_Quality_System_Monitor

Status STOPPED_BY_COMMAND

Stmt. # 147

Start Date/Time 4/25/99 12:34:56

End Date/Time 7/14/99 1:01:01

```
147  END WHEN
148  SET SPEED OF PUMP1 TO 100
149  SET SPEED OF PUMP2 TO 100
150  WHEN SPEED OF PUMP1 > 90 AND
150     SPEED OF PUMP2 > 90
151    WITHIN 100
152       COMMAND OWQS_MON CLARITY_TEST
153    OTHERWISE
154       STOP OPTICAL_WATER_QUALITY_SYS_MON
155  END WHEN
156  WAIT 10
157  IF CLARITY OF POTABLE_WATER < 4.5
158        COMMAND OWQS_MON PURGE, RATE => 25
```

data output fields

data input field

general action buttons

# Structure of UIL Executor Object Class

2/92

- Attributes:
    - NUMBER_OF_BUNDLES
    - BUNDLE _NAME
    - BUNDLE _STATUS
    - BUNDLE _DT
    - BUNDLE _PRIO
    - BUNDLE _N_SEQS
    - BUNDLE _USER_INFO
    - SEQUENCE_NAME
    - SEQUENCE_STATUS
    - SEQUENCE_START_TIME
    - SEQUENCE_STOP_TIME
    - SEQUENCE_STATEMENT
    - SEQUENCE_USER_INFO
    - UIL_DISPLAY_TEXT

Actions:

apply to "procedure"

- INSTALL
- REMOVE
- ABORT

apply to "sequence"

- START
- STOP
- PROCEED
- STEP
- HOLD_AT
- JUMP_TO

- Packaging of attributes:
    - One "UIL Executor" object, and/or
    - Object for each bundle executor

- Bundles (and sequences) are not objects -- bundles are files stored on the MSU

# Use as ITVE Tool

- TIMELINER has been used for 10 years at Draper Laboratory as a verification tool

- Available today for use with any one-machine, functional or real-time Ada simulation

- Scripts can provide the following capabilities:
  - Initialization
  - Injection of errors, perturbations
  - Simulation of human inputs
  - *Ad hoc* recovery of information
  - Avoidance of Ada recompilations

- ITVE version interfaces with sim via variables -- bypasses DMS
  - User provides "variable list" that describes variables

# Memory Sizing

2/92

- Used Verdix Compiler, version 6.0, target 80386
- Exec-time (onboard) processing:
  - Exact total 2650 stmts. 174312 bytes
  - Breakdown
    - Executor code 86000 bytes
    - Table of variables 60000 bytes
    - Buffers containing IL 28000 bytes
- Memory sensitive to number of variables, sizing of buffers, i.e. maximum size of installed scripts
- Sizes given above are based on 1087 variables, 1024 statements, 64 seqs/subseqs
- For SSF, remove or shrink variable list, add RODB, ISE interfaces
- Total memory required, exclusive of buffers: 100000 bytes

DRAPER LABORATORY

# CPU Performance

- Used Verdix Compiler, version 6.0, target 80386 at 20 Mhz.

- Timing for passes on which sequence is waiting for condition:

| | |
|---|---|
| WAIT (literal or variable) interval | 0.1 ms. |
| WHEN(EVER) not boolean | 0.4 ms. |
| WHEN(EVER) boolean = boolean | 0.7 ms. |
| WHEN(EVER) numeric > literal | 1.4 ms. |
| WHEN(EVER) numeric = numeric | 2.1 ms. |
| WHEN(EVER) numeric > literal<br>BEFORE boolean /= boolean | 2.1 ms. |
| WHEN(EVER) numeric > literal<br>WITHIN (literal or variable) interval | 1.5 ms. |

- Further refinement will improve time consumption

- Several hundred sequences can be processed at once

# Comparison of UIL / ILE Options

2/92

| | | | |
|---|---|---|---|
| object-oriented features | | | X |
| goto statement | | | X |
| for statement | | | X |
| return statement | | | X |
| user-defined functions | | | X |
| extensible commands, objects, classes | | | X |
| conditional logic (TL: when, if; UIL: verify, choice) | | X | X |
| event handling (TL: whenever; UIL: event handler) | | X | X |
| loop capability (TL: every; UIL: loop, for) | | X | X |
| parallel strings of execution | | X | X |
| call capability | | X | X |
| user control | X | X | X |
| delta-time commands, i.e. "wait" | X | X | X |
| time-tagged commands | X | X | X |
| RODB attribute read | | X | X |
| RODB attribute write | | X | X |
| RODB action command | X | X | X |
| MEMORY (k-bytes not including buffers) | 57 | 100 | 208 |
| COST (x $1,000,000) | 1.74 | 1.1 | 5-10 |
| | 2B | TL | UIL |

| | |
|---|---|
| Opt 2B | IBM Option 2B "time-tag action list with limited user control" |
| TL | Draper Laboratory TIMELINER |
| UIL | IBM Option 3 "UIL with conditional logic and internal variables" |

new

# Cost and Schedule

- Assumption: continuing support for DMS test-bed simulation to be used for interface development and verification

- Deliverables:
    - Documentation: FSSR, design documents, user's guide
    - System releases:
        - Demonstrator (immediate) -- can be interfaced with any one-machine Ada simulation via variables -- useful for language familiarization -- also useful as ITVE tool -- user's guide available
        - Alpha (6 months) -- early release
        - Beta (12 months) -- mature release
        - Final (18 months) -- fully-verified

- Further testing on DMS kit when kit available

- Cost: about 6 man-years -- $1,100,000

# Conclusions

- TIMELINER is
  - Mature (10 years)
  - Reuses code already paid for by NASA
  - Almost as capable as full UIL
  - Satisfies needs of MOD, ISE, payloads, ITVE
  - Schedule supports program
  - Less costly than less capable alternatives
  - CSDL has test-bed suitable for development and verification

# Conclusions

- The core of the TIMELINER language is a set of constructs that facilitate the writing of sequencing logic
- Interfaces
- Existing flavors are
  - Simulation version that interfaces via sim variables
    - Single machine (functional) version
    - Multiple machine version -- uses Multibus II
  - SSF version that interfaces via RODB objects, attributes and actions

# Agenda

- This is the TIMELINER "kick-off"
- Big pitch — DE and PR
- Development flow
- Major task segments
  - MODB/RODB interface
  - Memory management and tasking
  - Command handling
  - Maintain Sim version

# AN INTRODUCTION TO TIMELINER

May 24, 1991

Don Eyles

(617) 258-2460

The Charles Stark Draper Laboratory, Inc.
Cambridge, MA 02139

# Problem 1 -- Simulation

5/24/91

- Need to
  - Initialize
  - Inject errors, perturbations
  - Simulate human inputs
  - Run a series of cases
  - Recover *ad hoc* information
  - Avoid recompilations, relinks

# Problem 2 -- Flight System

5/24/91

- Need to
  - Lower crew workload
  - Provide "integrated" executive function
  - Automate procedures
    - Routine spacecraft operation
    - Payload operations
    - Failure recovery
  - Sequence day/week/month activity
  - Provide monitoring capability

# Solution

- A language for writing procedures
- Specialized for sequencing
  - Time-oriented
  - Closed-loop, i.e. reactive
  - Parallel activities
  - English-like to facilitate  monitoring

# History

- HAL version, about 1981
  - Used by "integrated package", OFS20, and descendants
  - Scripts currently in use have over 2000 statements organized into over 100 sequences (Peter Kachmar)
- Fortran version, about 1986
  - Used by AFE Fortran simulation
- Ada version in 1991 for SSF real-time testbed
  - Functional and real-time multi-processor versions
  - Version with additional capability under development

# Concept

- **TIMELINER development based on paradigms:**

Ⓐ

```
WHEN  condition
   action
   action
WAIT  interval
   action
   action
WHEN  condition
   action
   action
etc....
```

Ⓑ

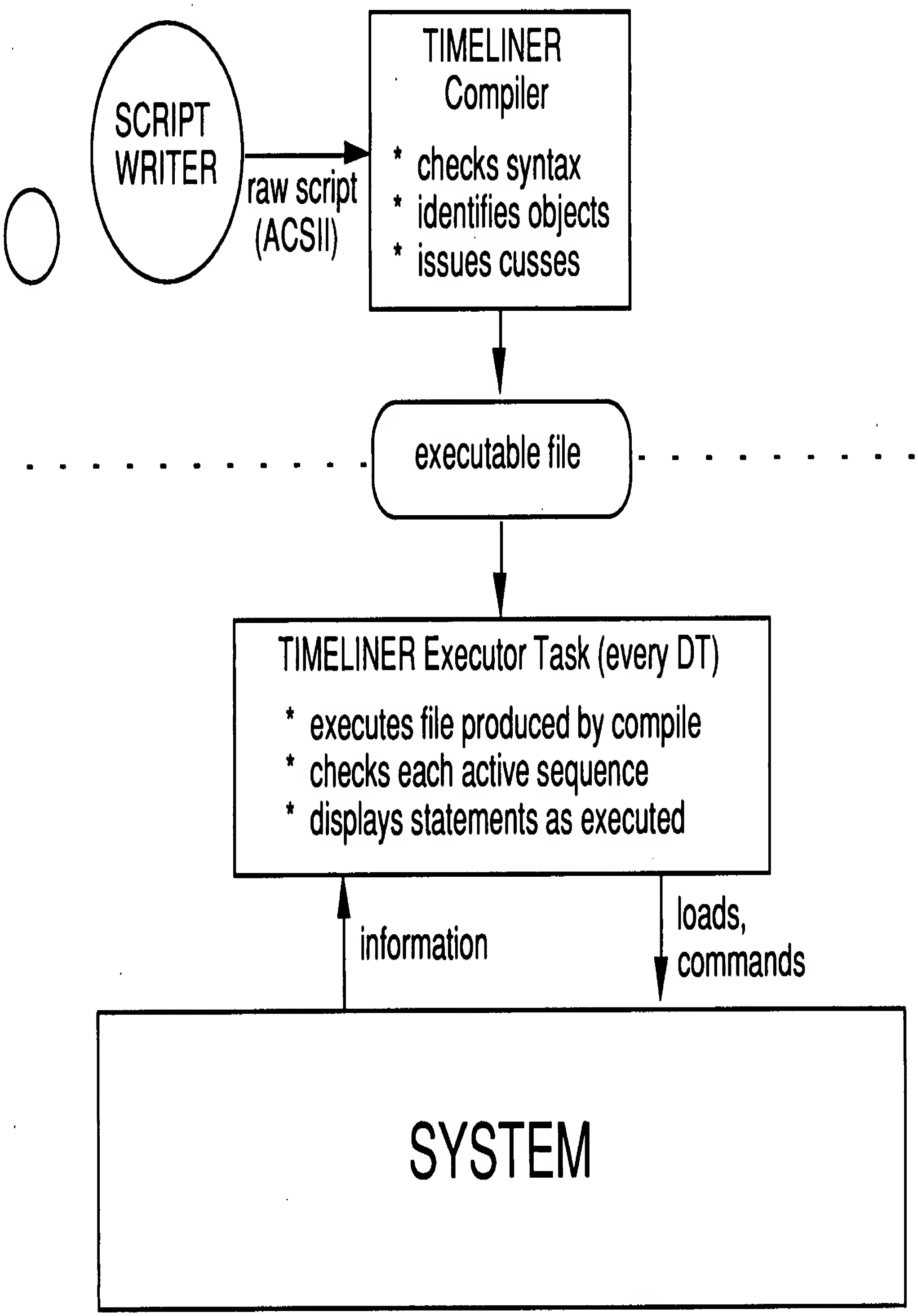
```
WHENEVER  condition
    action
    action
```

- **Multiple sequences, of either type, must perform in parallel**

- **"Condition" must be general**

# Structure

5/24/91



SCRIPT WRITER

raw script (ACSII)

TIMELINER Compiler

- checks syntax
- identifies objects
- issues cusses

executable file

TIMELINER Executor Task (every DT)

- executes file produced by compile
- checks each active sequence
- displays statements as executed

information

loads, commands

SYSTEM

# Script Organization

5/24/91

- Scripts composed of multiple "sequences" and "subsequences"
- Serial / parallel processing of sequences within script
  - Each sequence operates serially
  - Sequences operate in parallel with each other
  - Typical sequences are
    - Long -- to sequence mission (WHENs and WAITs)
    - Short -- to handle special conditions (WHENEVER)
  - Subsequences similar to sequences except
    - Do not establish independent stream of execution
    - Executed in-line as part of sequence that calls them